On Flexible Dynamic Trait Replacement for Java-like Languages

Published version:

DOI:10.1016/j.scico.2012.11.003

On Flexible Dynamic Trait Replacement for JAVA-like Languages[☆]

Lorenzo Bettini[*,a], Sara Capecchi[a], Ferruccio Damiani[a]

[a] *Dipartimento di Informatica, Università di Torino*

Abstract

Dynamic trait replacement is a programming language feature for changing the objects' behavior at runtime by replacing some of the objects' methods. In previous work on dynamic trait replacement for JAVA-like languages, the objects' methods that may be replaced must correspond exactly to a named trait used in the object's class definition. In this paper we propose the notion of *replaceable*: a programming language feature that decouples trait replacement operation code and class declaration code, thus making it possible to refactor classes and to perform unanticipated trait replacement operations without invalidating existing code. We give a formal account of our proposal through a core calculus, FDTJ (FEATHERWEIGHT DYNAMIC TRAIT JAVA), equipped with a static type system guaranteeing that in a well-typed program no runtime type error will take place.

Key words: Featherweight Java, Trait, Type System

1. Introduction

The term *trait* was used by Ungar et al. [43, 42], in the dynamically-typed prototype-based language SELF, to describe a parent object to which an object may delegate some of its behavior. Subsequently, Schärli et al. [40, 18] introduced traits in the dynamically-typed class-based language SQUEAK/SMALLTALK as a mechanism for fine-grained reuse. A trait is a set of methods completely independent from any class hierarchy. Traits can be composed to form new traits or classes and trait composition seems to provide a more flexible support to code reuse than (single and multiple) class-based inheritance. They can be composed in an arbitrary order and require the composed trait or class to resolve possible name conflicts explicitly. These features make traits more flexible and semantically simpler than *mixins* [10, 27, 20, 4]. Because of their flexibility and simple semantics, traits have attracted a great deal of attention and various formulations of traits within a nominal JAVA-like type system can be found in the literature (see, e.g., [41, 31, 28, 37, 9]). A form of trait construct is present also in the recent programming language FORTRESS [3] (where there is no class-based inheritance), while the "trait" construct incorporated in SCALA [32] is indeed a form of mixin.

Dynamic trait replacement is the ability to change the behavior of an object at runtime by replacing one trait for another. (Dynamic trait replacement was listed as an issue for further work in the papers on traits in the dynamically-typed language SQUEAK/SMALLTALK [40, 18]). In the prototype-based language SELF dynamic trait replacement can be achieved by changing the reference to the parent of an object. In the class-based setting, dynamic trait replacement has been formalized by Smith and Drossopoulou through the JAVA-like language *Chai*$_3$ [41]. The language *Chai*$_3$ contains an operator for replacing a trait in an existing object. This operator requires that the trait to be replaced corresponds exactly to a named trait used in the object's class definition. That is, the sets of methods that may be dynamically replaced are fixed in the object's class definition and the trait replacement operations are coupled to the names of the traits used in the object's class definition.

[☆] This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

[*] Corresponding Author.

Email addresses: bettini@di.unito.it (Lorenzo Bettini), capecchi@di.unito.it (Sara Capecchi), damiani@di.unito.it (Ferruccio Damiani)

In this paper we propose a more flexible dynamic trait replacement operator. A distinguishing feature of our proposal is the notion of *replaceable*, a feature that provides a means to decouple trait replacement operation code and class declaration code, thus enabling class refactoring and unanticipated trait replacement operations without invalidating existing code. We give a formal account of our proposal through a core calculus, FDTJ (FEATHERWEIGHT DYNAMIC TRAIT JAVA), equipped with a static type system guaranteeing that in a well-typed program no runtime type error will take place. A preliminary version of the material presented in this paper appeared as [7].

Organization of the paper. Section 2 presents some background and motivation. Section 3 illustrates our proposal through examples. Section 4 presents syntax, type system and operational semantics of FDTJ, a minimal core calculus for dynamic trait replacement. Section 5 shows that the FDTJ static type system is sound: it ensures that in a well-typed program no runtime error will take place. Section 6 discusses some related work. We conclude by summarizing the paper and outlining possible directions for further work.

2. Background and Motivation

The only dynamic feature in JAVA-like languages is represented by dynamic binding, i.e., the dynamic selection of a specific method implementation according to the runtime type of an object. This is not enough for representing the dynamic evolution of objects that behave differently depending on their internal state, the context where they are executing or the entities they interact with. All these possible behaviors may not be completely predictable in advance and they are likely to change after the application has already been developed and used. While trying to forecast all the possible evolutions of system entities, classes are often designed with too many responsibilities, most of which are basically not used. Furthermore, the number of subclasses tend to grow dramatically when trying to compose different functionalities into single modules.

Design patterns [21] are a programming techniques enabling to overcome some problematic deficiencies present in class based object-oriented programming languages. For instance to extend and change dynamic objects behavior at runtime in a class based context we can use *Decorator* and *State* pattern respectively. Design patterns are useful, however they require manual programming (decreasing productivity). The programmer should explicitly comment the code about the patterns used and their correct usage, but this still remains an informal specification, and most of the responsibility is left to the user of such implemented components. On the contrary, linguistic extensions also "document" the code [13]. For instance, the replaceable construct can be seen as an abstraction of objects' roles/modes that directly highlights the main concept underneath its use (see, e.g., RPolicy in Listing 10 and RClassification in Listing 12). The correct use of the new linguistic constructs can be checked statically and once proved correct, their usage is guaranteed, at runtime, not to produce any runtime type errors.

The replaceable construct aims to achieve, in a class-based statically-typed setting, part of the dynamic flexibility typical of prototype-based languages [43, 42]. The same goal is shared, for instance, by the various proposals for statically typed delegation features than can be found in the literature [25, 33, 35, 6] (we also refer to the Related Work, Section 6). A distinguished feature of the replaceable construct is that it smoothly integrates into class based languages that already support traits.

2.1. An Example

Consider an application handling bank accounts, represented by objects of class CAccount sketched in Listing 1, where the interface IAccount lists all the public methods of account objects. Suppose that the application has to evolve to support the unanticipated need to classify bank accounts as Gold, Standard or Bad according to their reliability. A possible way of supporting the new feature is to rewrite the class CAccount by exploiting the design pattern State [21], which is often used in contexts where it is needed to change object behaviors at runtime according to their state.

The evolved code is sketched in Listing 2, where the interface IAccountState, the classes CStandardAccount, CBadAccount, CGoldAccount and CAccount represent a possible implementation using the State design pattern. The interface IAccountState includes methods calculateInterests used to calculate interests (according to the balance and the class of the customer) and sendSummary sending to the customer's address a monthly summary of the operations made on the account. Class CAccount includes an attribute of type IAccountState used to switch the state of the current account through method changeState. The execution of methods belonging to the interface IAccountState

```
interface  IAccount {
   double getBalance();
   double calculateInterests();
   void sendSummary();
}

class CAccount implements IAccount {
   double balance;
   Address a;

   CAccount(...) {...}

   double getBalance() { return balance; }
   double calculateInterests() {...}
   void sendSummary() {...}
}
```

Listing 1: The account example

are delegated to the attribute state that, thanks to dynamic binding, performs the body of the method associated to the current state of the account. The state can change after a check of the associated balance as in the following code:

```
void checkCustomer(IAccount c) {
   if  (c.getBalance()<0)  { c.changeState(new CBadAccount()); }
   else if  (c.getBalance()>1000) { c.changeState(new CGoldAccount()); }
}
```

Unfortunately, the above solution, that required a significant amount of manual programming, does not scale well to unanticipated extensions. Suppose, for instance, that a new, unanticipated category of customers has to be modeled: customers can decide to switch to an online-only account which is less expensive and does not send the monthly summary. The problem is that this feature is independent from the customers' state: online customers can still be Gold, Standard and Bad. The hierarchy in Listing 2 has to be extended by adding the three following subclasses (where OL stands for online):

```
class COLGoldAccount extends CGoldAccount { void sendSummary(Address a) {} }
class COLStandardAccount extends CStandardAccount { void sendSummary(Address a) {} }
class COLBadAccount extends CBadAccount { void sendSummary(Address a) {} }
```

Then, when transforming an account in a online one we have to check its state to find the right subclass:

```
void setOnline(CAccount b){
   if (b instanceof CGoldAccount) { c.changeState(new COLGoldAccount()); }
   else if (b instanceof CBadAccount) { c.changeState(new COLBadAccount()); }
   else if (b instanceof CStandardAccount) { c.changeState(new COLStandardAccount()); }
}
```

This solution based on the State pattern is not so flexible w.r.t. unanticipated extensions since the combination of new features with preexisting states causes subclasses explosion; moreover the code to change the state (as the setOnline method above) is cumbersome and error prone in case of multiple combinations of state-dependent features.

To avoid the explosion of subclasses, one could use the decorator pattern [21] together with the state pattern: this way the online state would be a decorator of the other states. However, this requires additional programming, and might also require to refactor the state hierarchy in order to take in consideration also decorators. Indeed, the OnLine decorator class would have a reference to a bank account, would redefine the method sendSummary as empty and would forward all the other methods to the bank account reference. Furthermore, if the bank account changes in the future, then also the decorator classes will have to be adapted.

The instanceof tests could be avoided implementing setOnline or changeState methods as *multi-methods* or in general using *dynamic overloading* if the language supports these mechanisms (see, e.g., [16, 30, 12, 39, 14, 8]), that

```
interface  IAccountState {
   double calculateInterests(double d);
   void sendSummary(Address a);
}

class CGoldAccount implements IAccountState {
   double calculateInterests(double d) {...}
   void sendSummary(Address a) {...}
}

class CStandardAccount implements IAccountState {
   double calculateInterests(double d) {...}
   void sendSummary(Address a) {...}
}

class CBadAccount implements IAccountState {
   double calculateInterests(double d) {...}
   void sendSummary(Address a) {...}
}

interface  IAccount {
   void changeState(IAccountState s);
   double getBalance();
   double calculateInterests();
   void sendSummary();
}

class CAccount implements IAccount {
   double balance;
   Address a;
   IAccountState state;

   CAccount(...) {...}

   void changeState(IAccountState s) { state = s; }
   double getBalance() { return balance; }
   double calculateInterests() { return state.calculateInterests(balance); }
   void sendSummary() { state.sendSummary(a); }
}
```

Listing 2: The evolved account example using the State pattern

```
interface ISequence {
   void in(Object o);  // inserts an element
   void out();  // removes an element
   Object get();  // returns an element without removing it
   boolean isEmpty();  /* checks whether this sequence is empty */ }

trait TFifo is {
   List l;  // required field
   void in(Object o) { l.addFirst(o); }  // provided method
   void out() { l.removeLast(); }  // provided method
   Object get() { return l.getLast(); }  // provided method
   boolean isEmpty() { return (l.size() == 0); }  /* provided method */ }
```

Listing 3: The interface ISequence and the trait TFifo

is, runtime method selection mechanism based on the dynamic types of the receiver and the arguments. However, any time we want to add a new state to the hierarchy we also have to add the corresponding multi-method branch in the class where the multi-method is defined.

In Section 3.4 we will show how, in a language that supports traits, the replaceable construct proposed in this paper could be used to support these unanticipated software evolution examples.

3. Introducing Traits, Dynamic Trait Replacement and Replaceables

In Sections 3.1, 3.2 and 3.3 we introduce the notions of trait, dynamic trait replacement and replaceable through simple examples about data structures for sequences. Then in Section 3.4 we code the account example of Section 2.1 using our new language constructs. In the examples we exploit a JAVA-like notation and use a more general syntax (including, e.g., the types void and int, the assignment operator, etc.) than the one of the calculus that we will present in Section 4.

3.1. Introducing Traits

Consider the development of a class FifoSequence, by means of a First-in-first-out policy, implementing the interface ISequence in Listing 3.

In a language with traits the class FifoSequence can be developed by first introducing a trait TFifo providing the methods (see Listing 3) and then composing the class, by exploiting the trait, as follows

```
class FifoSequence implements ISequence by TFifo {
   List l; // provided field
   FifoSequence() { this.l = new LinkedList(); } // constructor
}
```

Subsequently, a class LifoSequence implementing the ISequence interface by means of a Last-in-first-out policy can be developed by reusing the methods in and isEmpty provided by the trait TFifo to define the trait TLifo in Listing 4 (the operation exclude forms a new trait by excluding a method from a given trait and the sum operation, +, merges two traits to form a new trait — see [18, 9]). The class LifoSequence can be defined as follows

```
class LifoSequence implements ISequence by TLifo {
   List l;
   LifoSequence() { this.l = new LinkedList(); }
}
```

```
trait TLifo is (TFifo[exclude out][exclude get])
            + { List l; // required field
                void out() { l.removeFirst(); } // provided method
                Object get() { return l.getFirst(); } /* provided method */ }
```

Listing 4: The trait TLifo

```
class Sequence implements ISequence by TFifo {
   List l;
   Sequence() { this.l = new LinkedList(); }
}
```

Listing 5: The class Sequence

3.2. Introducing Dynamic Trait Replacement

The classes FifoSequence and LifoSequence illustrate the use of traits as a static construct: the composition of traits into a class is fixed, once and for all, at compile time. For an example illustrating dynamic trait replacement consider the problem of developing a method changePolicy that takes as argument an instance s of the class Sequence of Listing 5, changes the extraction policy of s, extracts an element from s (according to the new policy), and terminates by changing further the insertion/extraction policy of s.

The Lexicographical-ordering extraction policy (that extracts elements from a sequence according to lexicographical ordering on the values returned by invoking the toString method on its elements) and the First-in-first-out insertion/extraction policy can be described by the traits TLoExtractionPolicy and TFifoPolicy in Listing 6, respectively.

During the execution, the extraction policy of the sequence s can be changed into Lexicographical-ordering by means of the trait replacement operation s{TLoExtractionPolicy}. Once the replacement has taken place, invoking the methods get and out will select the implementation of such methods as provided by TLoExtractionPolicy. The insertion/extraction policy can be then changed to First-in-first-out by executing the operation s{TFifoPolicy}. This is illustrated in the following code:

```
void changePolicy(Sequence s) {
   s{TLoExtractionPolicy};   // (1)
   System.out.println(s.get());
   s.out();
   s{TFifoPolicy}   // (2)
}
```

Note that there is no coupling between the definition of the class Sequence and the trait replacement operations. Consider, for instance, the need to develop the methods disableIn and enableIn that take a sequence object s and disable/enable the in operation without affecting the get and out operations. The methods disableIn and enableIn can be written, without changing the code of the class Sequence, by first defining the traits TinDisabled and TinEnabled in Listing 7. Then, the code of the methods is as follows

```
void disableIn(Sequence s) {
   s{TinDisabled};   // (3)
}
void enableIn(Sequence s) {
   s{TinEnabled};   // (4)
}
```

Since in the proposed approach there is no coupling between the definition of a class and the trait replacement operations on its instances, any change to the code of class Sequence that preserves the signatures of its methods does not affect existing trait replacement operations. For instance, the trait replacement operations (1), (2), (3) and (4) that have been written to deal with the class Sequence in Listing 5 are still valid when the class is rewritten, as in Listing 8, in order to make sequences to be created as Last-in-first-out sequences.

```
trait TLoExtractionPolicy is {
   List l;  // required field
   void out() { ... }  // provided method (removes the minimum element according to Lexicographical−ordering)
   Object get() { ... }  /* provided method (returns the minimum element according to Lexicographical−ordering) */ }

trait TFifoPolicy is {
   List l;  // required field
   void in(Object o) { l.addFirst(o); }  // provided method
   void out() { l.removeLast(); }  // provided method
   Object get() { return l.getLast(); }  /* provided method */ }
```

Listing 6: The traits TFifoPolicy and TLoExtractionPolicy

```
trait TinDisabled is { void in(Object o) { }  /* provided method */ }

trait TinEnabled is { void in(Object o) { l.addFirst(o); }  /* provided method */ }
```

Listing 7: The traits TinDisabled and TinEnabled

3.3. Introducing Replaceables

Consider the problem of developing a more general version of the method changePolicy of Section 3.2, let us call it changePolicy1, that can accept as argument an object s belonging to any class that implements the interface ISequence. Unfortunately, the signature

```
void changePolicy1(ISequence s)
```

is not able to guarantee at compile time that the dynamic replacement of the methods in, out and get of s is type safe. In fact it is not enough to simply check that the replacing methods have the same signatures as the replaced ones: these methods may rely on other methods and/or fields. In order to be able to statically type-check runtime method replacement when the static type of the target expression e of the replacement operation $\mathsf{e}\{\mathsf{T}\}$ is not a class, we introduce the notion of *replaceable* as mean to specify these requirements.

A replaceable is a predicate over a set of methods (that is, over a trait), declared independently from any interface hierarchy and from any other replaceable declaration. The syntax of a *replaceable definition* is as follows

$$\textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}};\mid\overline{\mathsf{Z}};\mid\overline{\mathsf{J}};\rangle$$

where $\overline{\mathsf{S}}$ and $\overline{\mathsf{Z}}$ are disjoint sequences of method signatures ($\overline{\mathsf{S}}$ are the methods they may be replaced and $\overline{\mathsf{Z}}$ are the methods that may be used by the methods in $\overline{\mathsf{S}}$ and may not be replaced), $\overline{\mathsf{G}}$ is a sequence of fields (the fields that may be used by the methods $\overline{\mathsf{S}}\cup\overline{\mathsf{Z}}$) and $\overline{\mathsf{J}}$ is a sequence of interface names (the interfaces that may be used as types of this within the bodies of the methods $\overline{\mathsf{S}}\cup\overline{\mathsf{Z}}$).

Interface names and replaceable names are used to form a novel kind of source language type that contains the information needed for statically type-checking runtime method replacement. Namely, given an interface I and a replaceable

$$\textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}};\mid\overline{\mathsf{Z}};\mid\overline{\mathsf{J}};\rangle$$

I{R} is the type of references to any object whose class implements the interface I and has the methods described by the replaceable R. Through a reference of type I{R} it is possible

- to invoke on the referenced object the methods of the interface I, and
- to replace, in the referenced object, any of the methods with signatures $\overline{\mathsf{S}}$ by any trait that satisfies R.

Note that, with respect to method invocation, the type I{R} is equivalent to the standard JAVA type I. The methods with signatures $\overline{\mathsf{Z}}$ may not be replaced. The signatures $\overline{\mathsf{Z}}$ provide, together with the fields $\overline{\mathsf{G}}$ and the interfaces $\overline{\mathsf{J}}$, additional constraints needed to ensure safety. Namely, to ensure safety, the class of the referenced object must:

```
class Sequence implements ISequence by TLifo {
   List l;
   Sequence() { this.l = new LinkedList(); }
}
```

Listing 8: A Last-in-first-out version of the class Sequence

```
replaceable RExample is { void outTwice(); boolean isNotEmpty(); boolean test(); }
                          < ISomeSequence s; ISomeSequenceComparator c;
                          | boolean isEmpty(); void out();
                          | ISomeSequence; >

interface ISomeSequence extends ISequence { void outTwice(); boolean isNotEmpty(); boolean test(); }

interface ISomeSequenceComparator { boolean compare(ISomeSequence s1, ISomeSequence s2); }

trait TExample1 is { ISomeSequence s;  // required field
                     ISomeSequenceComparator c;  // required field
                     void out();  // required method
                     boolean isEmpty();  // required method
                     void outTwice() { out(); out(); }  // provided method
                     boolean isNotEmpty() { return (! isEmpty()); }  // provided method
                     boolean test() { return c.compare(this,s); }  /* provided method */ }

trait TExample2 is { void out();  // required method
                     void outTwice() { out(); out(); }  /* provided method */ }
```

Listing 9: The replaceable RExample, the interfaces ISomeSequence, ISomeSequenceComparator and the traits TExample1, TExample2

- implement all the interfaces $\mathsf{I} \cup \overline{\mathsf{J}}$;
- have all the fields $\overline{\mathsf{G}}$; and
- have all the methods $\overline{\mathsf{S}} \cup \overline{\mathsf{Z}}$.

A trait T (that is, a set of methods) *satisfies* the replaceable R if and only if T consists of methods whose signatures occur in $\overline{\mathsf{S}}$ and whose bodies

- select only the fields $\overline{\mathsf{G}}$ on this,
- select only the methods $\overline{\mathsf{S}} \cup \overline{\mathsf{Z}}$ on this,
- assume only the interfaces $\overline{\mathsf{J}}$ as nominal types of this, i.e.,
  - pass this as argument to a method only if there exists an interface in $\overline{\mathsf{J}}$ that is a subtype of the type of the corresponding formal parameter of the method, and
  - return this as result of a method only if there exists an interface in $\overline{\mathsf{J}}$ that is a subtype of the return type of the method.

For instance, let us consider the replaceable RExample, the interfaces ISomeSequence and ISomeSequenceComparator and the traits TExample1 and TExample2 in Listing 9: RExample is satisfied by both TExample1 and TExample2.

Let EmptyR be a distinguished name for the replaceable defined by **replaceable** EmptyR **is** $\{\bullet\}\langle\bullet \mid \bullet \mid \bullet\rangle$, which is satisfied only by the empty set of methods. According to the above description, for every interface I, we will identify the types I$\{$EmptyR$\}$ and I.

```
replaceable RPolicy is { void out(); Object get(); void in(Object o); } < List l; | | >
```

Listing 10: The replaceable RPolicy

Remark 3.1. *While presenting the type system we will introduce the* specificationof *operator, which automatically extracts a replaceable definition from a trait (see Definition 4.12). In a full language, this operator would relieve programmers of the burden of writing all the details of replaceable declarations. For instance, the definition of the replaceable* RExample *in Listing 9 could also be written as:* **replaceable** RExample **is** specificationof(TExample1).

Now we can go back to the problem of developing the changePolicy1 method. Consider the replaceable RPolicy in Listing 10. According to the above explanation, a variable s of type ISequence{RPolicy} can be assigned an instance of any class that implements the interface ISequence and has the methods described by the replaceable RPolicy (like the class Sequence in Listing 5). During the execution, the extraction policy of the sequence can be changed into Lexicographical-ordering by means of the trait replacement operation s{TLoExtractionPolicy}. Once the replacement has taken place, invoking the methods get and out will select the implementation of such methods as provided by TLoExtractionPolicy. The insertion/extraction policy can be then changed to First-in-first-out by executing the operation s{TFifoPolicy}. This is illustrated in the following code:

```
void changePolicy1(ISequence{RPolicy} s) {
   s{TLoExtractionPolicy};    // (1')
   System.out.println(s.get());
   s.out();
   s{TFifoPolicy}    // (2')
}
```

The replaceable RPolicy describes both a subset of the methods of any class whose instances may be referenced through a variable of type ISequence{RPolicy} and a superset of the methods of the traits used by the trait replacement operations (1') and (2'); and this is all that is required to make the above code correct. The variant of the methods disableIn and enableIn of Section 3.2 that works on arguments of type ISequence{RPolicy} is as follows

```
void disableIn1(ISequence{RPolicy} s) {
   s{TinDisabled};    // (3')
}
void enableIn1(ISequence{RPolicy} s) {
   s{TinEnabled};    // (4')
}
```

Note that the trait replacement operations (1'), (2'), (3') and (4') work with both the versions of the class Sequence given in Listings 5 and 8.

3.4. The Account Example

Let us come back to the example in Section 2.1. In a language with traits the code in Listing 1 would have been written as in Listing 11, where the trait TAccount provides all the methods of account objects.[1]

The unanticipated need to classify bank accounts as Gold, Standard or Bad according to their reliability can be addressed by adding the code in Listing 12. There are three additional traits: TStandardAccount, TBadAccount and TGoldAccount that implement reliability-dependent methods. The replaceable RClassification is used to define references of type IAccount{RClassification}: the referenced objects can replace the implementation of the methods calculateInterests and sendSummary. Now let us implement method checkCustomer introduced in Section 2.1:

```
void checkCustomer(IAccount{RClassification} c) {
   if  (c.getBalance() < 0)  c{TBadAccount};
   else if  (c.getBalance() >1000)  c{TGoldAccount};
}
```

[1] In a language with traits it is good practice not to define methods in the body of classes, in order to maximize the opportunity for reuse.

```
interface IAccount ... // As in Listing 1

trait TAccount is { double balance; Address a;
   double getBalance(){ return balance; }
   double calculateInterests() {...}
   void sendSummary() {...}
}

class CAccount implements IAccount by TAccount {
   double balance;
   Address a;

   CAccount(...) {...}
}
```

Listing 11: The account example using traits

```
trait TStandardAccount is TAccount[exclude getBalance]

trait TGoldAccount is { double balance;  Address a;
   double calculateInterests() {...}
   void sendSummary() {...}
}

trait TBadAccount is { double balance;  Address a;
   double calculateInterests() {...}
   void sendSummary() {...}
}

replaceable RClassification is { double calculateInterests(); void sendSummary(); }
   <  double balance; Address a;  | |  >
```

Listing 12: Additional code for the evolved account example using traits and replaceables

```
interface IAccount ... // as in Listing 1

trait TBaseAccount { double balance;
   double getBalance(){ return balance; }
}

trait TStandardAccount is { double balance;  Address a;
   double calculateInterests() {...}
   void sendSummary() {...}
}

trait TGoldAccount is ... // As in Listing 12

trait TBadAccount is ... // As in Listing 12

class CAccount implements IAccount by TBaseAccount + TStandardAccount {
   ... // As in Listing 11
}

replaceable RClassification is ... // as in Listing 12
```

Listing 13: A refactoring of the code of the evolved account example in Listings 11 and 12

The parameter of the method is of type IAccount{RClassification}. With respect to the implementation in Section 2.1 we just have to perform the replaceable operation c{TSomeAccount} instead of: (i) creating a new object representing the state, and (ii) assign it to the corresponding c field through the method changeState. Later, to model online accounts, we just have to add a trait implementing the new version of sendSummary:

```
trait TOnLine { void sendSummary() {} }
```

and then to declare the parameter of the setOnline method as IAccount{RClassification}:

```
void setOnline(IAccount{RClassification} c){
   c{TOnLine};
}
```

Using traits and replaceables we do not have subclass explosion and we do not have to use instanceof tests in the body of the method setOnline: we just have to write the replacement operation c{Online}. The benefits of our solution with respect to the State pattern can be summarized as in the following:

- the management of the state associated to an object, that is a dedicated field and a method to change it, is no more needed; and
- different sets of states depending on orthogonal features can be combined and added in a flexible way.

It is worth mentioning that the same unanticipated software evolution can be accomplished starting from the code in Listing 1, that could be seen as legacy code developed in a version of the language that does not support traits. The code to be added is essentially the same as in Listing 12, the only difference is that trait TStandardAccount has to be defined from scratch by introducing a duplication of the code of the methods calculateInterests and sendSummary.

Finally, Listing 13 illustrates a refactoring of the code of the evolved bank account in Listings 11 and 12. The trait TBaseAccount implements those methods that are independent from reliability (getBalance) while traits TStandardAccount, TBadAccount and TGoldAccount implements reliability-dependent methods. The class CAccount implements the behavior of standard accounts by traits TBaseAccount and TStandardAccount.

$$\begin{array}{lcll}
\mathsf{ID} & ::= & \textbf{interface}\ \mathsf{I}\ \textbf{extends}\ \bar{\mathsf{I}}\ \{\ \bar{\mathsf{S}};\ \} & \text{interfaces}\\
\mathsf{S} & ::= & \mathsf{U}\ \mathsf{m}\ (\bar{\mathsf{U}}\ \bar{\mathsf{x}}) & \text{method headers}\\
\mathsf{U} & ::= & \mathsf{I}\ \mid \mathsf{I}\{\mathsf{R}\} & \text{source language types}\\
\mathsf{RD} & ::= & \textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\bar{\mathsf{S}};\}\langle\bar{\mathsf{F}};\ {\scriptstyle|}\ \bar{\mathsf{S}};\ {\scriptstyle|}\ \bar{\mathsf{I}};\rangle & \text{replaceables}\\
\mathsf{F} & ::= & \mathsf{U}\ \mathsf{f} & \text{fields}\\
\mathsf{TD} & ::= & \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \mathsf{TE} & \text{traits}\\
\mathsf{TE} & ::= & \{\ \bar{\mathsf{F}};\ \bar{\mathsf{S}};\ \bar{\mathsf{M}}\ \}\ \mid \mathsf{T} & \text{trait expressions}\\
\mathsf{M} & ::= & \mathsf{S}\ \{\ \textbf{return}\ \mathsf{e};\ \} & \text{methods}\\
\mathsf{e} & ::= & \mathsf{x} \mid \mathsf{this.f} \mid \mathsf{e.m}(\bar{\mathsf{e}}) \mid \textbf{new}\ \mathsf{C}(\bar{\mathsf{e}})\ \mid \mathsf{e}\{\mathsf{T}\} & \text{expressions}\\
\mathsf{CD} & ::= & \textbf{class}\ \mathsf{C}\ \textbf{implements}\ \bar{\mathsf{I}}\ \textbf{by}\ \mathsf{TE}\ \{\ \bar{\mathsf{F}};\ \mathsf{K}\ \} & \text{classes}\\
\mathsf{K} & ::= & \mathsf{C}(\bar{\mathsf{U}}\ \bar{\mathsf{f}})\ \{\ \mathsf{this}.\bar{\mathsf{f}} = \bar{\mathsf{f}};\ \} & \text{constructors}
\end{array}$$

Figure 1: FDTJ Syntax

4. A Calculus for Dynamic Traits

In this section we introduce FDTJ (FEATHERWEIGHT DYNAMIC TRAIT JAVA), a minimal core calculus, in the spirit of FJ (FEATHERWEIGHT JAVA) [24], for interfaces, replaceables, traits, and classes. Since our goal is to provide a foundation for flexible dynamic trait replacement within a JAVA-like nominal type system, FDTJ does not model type casts, trait composition operations and class-based inheritance, which are orthogonal to dynamic trait replacement. Moreover, to further simplify the calculus, fields can be selected only on this.

The distinguishing design choices of FDTJ are the following.

- The novel *replaceable* construct specifies methods that can be dynamically replaced in an object by other methods that satisfy the specification. It provides a means to decouple trait replacement operation code and class declaration code, making it possible to refactor classes and performing unanticipated trait replacement operations without invalidating existing code. Replaceables are declared independently from interface declarations and from other replaceable declarations.

- The type system supports the typechecking of traits in isolation from the trait replacement operations that use them, so that it is possible to typecheck a method defined in a trait only once (instead of having to typecheck it in every trait replacement operation using that trait).

- Class names and trait names are not source language types. This choice allows us to simplify the calculus and to focus on the challenging problem of typing a dynamic trait replacement expression $\mathsf{e}\{\mathsf{T}\}$ when the (compile-time) type of the expression e is not a class (*Chai*$_3$ [41] does not consider interfaces).

Extending the calculus with type casts and trait composition operations is straightforward. The corresponding rules can be obtained by adapting those presented in [9], where dynamic trait replacement is not considered, to deal with the richer types and constraints introduced in this paper. Moreover, adding class-based inheritance and allowing the programmer to use class names and trait names as types should not pose particular technical problems (it would just complicate the calculus).

4.1. Syntax

The syntax of FDTJ is presented in Figure 1. We use the overbar sequence notation according to [24]. For instance, the pair "$\bar{\mathsf{U}}\bar{\mathsf{x}}$" stands for "$\mathsf{U}_1\,\mathsf{x}_1,\ldots,\mathsf{U}_n\,\mathsf{x}_n$", and "$\bar{\mathsf{U}}\bar{\mathsf{f}};$" stands for "$\mathsf{U}_1\,\mathsf{f}_1;\ldots;\mathsf{U}_n\,\mathsf{f}_n;$". The empty sequence is denoted by "•" and the length of a sequence $\bar{\mathsf{S}}$ is denoted by $|\bar{\mathsf{S}}|$. Sequences of named elements (interface definitions, method headers, method definitions, etc.) are assumed not to contain elements with the same name. Given a sequence of named elements $\bar{\mathsf{D}}$, the sequence of the names of the elements of $\bar{\mathsf{D}}$ is denoted by *names*$(\bar{\mathsf{D}})$, the subsequence of the elements of $\bar{\mathsf{D}}$ with the names $\bar{\mathsf{n}}$ is denoted by *choose*$(\bar{\mathsf{D}},\bar{\mathsf{n}})$, and *exclude*$(\bar{\mathsf{D}},\bar{\mathsf{n}})$ denotes the sequence obtained from

$\overline{\mathsf{D}}$ by removing the elements with the names $\overline{\mathsf{n}}$. We use a set-based notation for operators over sequences of named elements. In the union and in the intersection of sequences, denoted by $\overline{\mathsf{S}} \cup \overline{\mathsf{Z}}$ and $\overline{\mathsf{S}} \cap \overline{\mathsf{Z}}$, respectively, it is assumed that if $\mathsf{n} \in \mathit{names}(\overline{\mathsf{S}})$ and $\mathsf{n} \in \mathit{names}(\overline{\mathsf{Z}})$ then $\mathit{choose}(\overline{\mathsf{S}}, \mathsf{n}) = \mathit{choose}(\overline{\mathsf{Z}}, \mathsf{n})$. In the disjoint union of sequences, denoted by $\overline{\mathsf{S}} \cdot \overline{\mathsf{Z}}$, it is assumed that $\mathit{names}(\overline{\mathsf{S}}) \cap \mathit{names}(\overline{\mathsf{Z}}) = \bullet$.

A trait definition **trait** T **is** TE associates a trait name T to a trait expression TE. A trait expression is either a basic trait expression or a trait name. A basic trait expression $\{\,\overline{\mathsf{F}};\ \overline{\mathsf{S}};\ \overline{\mathsf{M}}\,\}$ defines methods $\overline{\mathsf{M}}$ and declares required fields $\overline{\mathsf{F}}$ and methods $\overline{\mathsf{S}}$ (where $\mathit{names}(\overline{\mathsf{S}}) \cap \mathit{names}(\overline{\mathsf{M}}) = \bullet$). Required fields and methods can be directly accessed (i.e., selected on this) within the bodies of the methods $\overline{\mathsf{M}}$.

A class definition **class** C **implements** $\overline{\mathsf{I}}$ **by** TE $\{\ \overline{\mathsf{F}};\ \mathsf{K}\ \}$ defines its fields, $\overline{\mathsf{F}}$, and the constructor, K, which shows how to initialize all the fields with the received arguments. The class does not directly define the methods of its interface $\overline{\mathsf{I}}$: it relies on a trait expression TE for this (with the clause **by**).

The subset of FDTJ obtained by removing the portions of the syntax highlighted in gray maps to a subset of JAVA. Namely, the FDTJ class **class** C **implements** $\overline{\mathsf{I}}$ **by** $\{\,\overline{\mathsf{F}};\ \bullet;\ \overline{\mathsf{M}}\,\}\ \{\ \overline{\mathsf{G}};\ \mathsf{K}\ \}$ (where the required field declarations $\overline{\mathsf{F}}$ are a subset of the provided field declarations $\overline{\mathsf{G}}$) can be understood as the JAVA class **class** C **implements** $\overline{\mathsf{I}}$ $\{\ \overline{\mathsf{G}};\ \mathsf{K}\ \overline{\mathsf{M}}\ \}$.

The expression $\mathsf{e}\{\mathsf{T}\}$ describes the dynamic replacement, in the object denoted by e, of the methods defined by the trait T for the corresponding methods of the object. For instance, if T is defined by **trait** T **is** $\{\,\overline{\mathsf{F}};\ \overline{\mathsf{S}};\ \overline{\mathsf{M}}\,\}$, then the methods of object e with names $\mathit{names}(\overline{\mathsf{M}})$ will be replaced by the methods $\overline{\mathsf{M}}$.

A replaceable definition **replaceable** R **is** $\{\overline{\mathsf{S}};\}\langle\overline{\mathsf{F}};\shortmid\overline{\mathsf{Z}};\shortmid\overline{\mathsf{I}};\rangle$ defines a replaceable of name R.

A class table CT is a map from class names to class declarations. Similarly, an interface table IT, a trait table TT, and a replaceable table RT map interface, trait, and replaceable names to interface, trait and replaceable declarations, respectively. An FDTJ program is a 5-tuple $(\mathsf{IT}, \mathsf{TT}, \mathsf{RT}, \mathsf{CT}, \mathsf{e})$, where e is the expression to be executed. Following FJ [24], in presenting the type system and the operational semantics we assume fixed, global tables IT, TT, RT, and CT. We also assume that these tables are *well-formed*, i.e., they contain an entry for each interface/trait/replaceable/class mentioned in the program, and the interface subtyping and trait reuse graphs are acyclic. In the following, instead of writing $\mathsf{CT}(\mathsf{C}) =$ **class** $\mathsf{C}\ \ldots$ we will simply write **class** $\mathsf{C}\ \ldots$; the same convention will be used for interfaces, replaceables and traits.

4.1.1. Lookup Functions and Method Signatures

In order to define FDTJ well-formed source language types, typing rules, and reduction rules we need a few lookup functions, given in Figure 2. The fields of a class C or required by a replaceable R are denoted by $\mathit{fields}(\mathsf{C})$ and $\mathit{fields}(\mathsf{R})$, respectively. Given a source language type $\mathsf{U} = \mathsf{I}\{\mathsf{R}\}$, we write $\mathit{fields}(\mathsf{U})$ to denote the fields required by R.

The interfaces implemented by the class C are denoted by $\mathit{interfaces}(\mathsf{C})$. Given a source language type $\mathsf{U} = \mathsf{I}\{\mathsf{R}\}$, we write $\mathit{interfaces}(\mathsf{U})$ to denote the interface name I, while $\mathit{allInterfaces}(\mathsf{U})$ denotes the sequence formed by the interface name I and all the interface names listed in the replaceable R.

The methods of a class C or of a trait expression TE are denoted by $\mathit{methods}(\mathsf{C})$ and $\mathit{methods}(\mathsf{TE})$, respectively.

Method *signatures*, ranged over by σ and ζ, are method headers deprived of parameter names. The signature associated to the method header S and the signature associated to the method definition M are denoted by $\mathit{mSig}(\mathsf{S})$ and $\mathit{mSig}(\mathsf{M})$, respectively. We write $\mathit{mSig}(\mathsf{C})$ and $\mathit{mSig}(\mathsf{I})$ to denote the signatures of all the methods of the class C and the signatures of all the methods of the interface I, respectively. Given a source language type $\mathsf{U} = \mathsf{I}\{\mathsf{R}\}$ we write $\mathit{mSig}(\mathsf{U})$ to denote the sequence containing the signatures of the interface I, the signatures of the method headers occurring in R and the signatures of the interfaces whose names are listed in R. Note that the lookup function $\mathit{mSig}(\cdot)$ is also defined on sequences.

Convention 4.1. *Since, in the method headers listed in a replaceable definition* **replaceable** R **is** $\{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}};\shortmid\overline{\mathsf{Z}};\shortmid\overline{\mathsf{J}};\rangle$, *the names of the parameters of the methods are immaterial, in the following we will sometimes write replaceable definitions by using method signatures instead of method headers.*

4.1.2. Well-Formed Source Language Types

A source language type U is well-formed if and only if the lookup $\mathit{mSig}(\mathsf{U})$ is defined, that is, there are no conflicts in the collected method signatures (recall that: sequences of method signatures are assumed not to contain

Fields lookup (function *fields*):

$$
\begin{array}{ll}
\mathit{fields}(\mathsf{C}) = \overline{\mathsf{F}} & \textbf{if class } \mathsf{C} \cdots \{\, \overline{\mathsf{F}};\ \mathsf{C}(\overline{\mathsf{F}})\{\cdots\}\,\} \\
\mathit{fields}(\mathsf{R}) = \overline{\mathsf{F}} & \textbf{if replaceable } \mathsf{R} \textbf{ is } \{\cdots\}\langle\overline{\mathsf{F}};\mid\cdots\mid\cdots\rangle \\
\mathit{fields}(\mathsf{I}\{\mathsf{R}\}) = \mathit{fields}(\mathsf{R}) &
\end{array}
$$

Interfaces lookup (functions *interfaces* and *allInterfaces*):

$$
\begin{array}{ll}
\mathit{interfaces}(\mathsf{C}) = \overline{\mathsf{I}} & \textbf{if class } \mathsf{C} \textbf{ implements } \overline{\mathsf{I}} \textbf{ by } \cdots \\
\mathit{interfaces}(\mathsf{I}\{\mathsf{R}\}) = \mathsf{I} & \\
\mathit{allInterfaces}(\mathsf{I}\{\mathsf{R}\}) = \mathsf{I} \cup \overline{\mathsf{J}} & \textbf{if replaceable } \mathsf{R} \textbf{ is } \{\cdots\}\langle\cdots\mid\cdots\mid\overline{\mathsf{J}};\rangle
\end{array}
$$

Methods lookup (function *methods*):

$$
\begin{array}{ll}
\mathit{methods}(\mathsf{C}) = \mathit{methods}(\mathsf{TE}) & \textbf{if class } \mathsf{C} \cdots \textbf{ by } \mathsf{TE}\,\{\,\cdots\,\} \\
\mathit{methods}(\mathsf{T}) = \mathit{methods}(\mathsf{TE}) & \textbf{if trait } \mathsf{T} \textbf{ is } \mathsf{TE} \\
\mathit{methods}(\{\cdots;\ \cdots;\ \overline{\mathsf{M}}\}) = \overline{\mathsf{M}} &
\end{array}
$$

Method signatures lookup (function *mSig*):

$$
\begin{array}{ll}
\mathit{mSig}(\mathsf{U}\,\mathsf{m}(\mathsf{U}_1\,\mathsf{x}_1,\dots,\mathsf{U}_n\,\mathsf{x}_n)) = \mathsf{U}\,\mathsf{m}(\mathsf{U}_1,\dots,\mathsf{U}_n) & \\
\mathit{mSig}(\mathsf{S}_1\dots\mathsf{S}_n) = \mathit{mSig}(\mathsf{S}_1)\cdot\dots\cdot\mathit{mSig}(\mathsf{S}_n) & \\
\mathit{mSig}(\mathsf{S}\,\{\textbf{return } \mathsf{e};\}) = \mathit{mSig}(\mathsf{S}) & \\
\mathit{mSig}(\mathsf{M}_1\dots\mathsf{M}_n) = \mathit{mSig}(\mathsf{M}_1)\cdot\dots\cdot\mathit{mSig}(\mathsf{M}_n) & \\
\mathit{mSig}(\mathsf{C}) = \mathit{mSig}(\mathit{methods}(\mathsf{C})) & \\
\mathit{mSig}(\mathsf{C}_1,\dots,\mathsf{C}_n) = \mathit{mSig}(\mathsf{C}_1)\cdot\dots\cdot\mathit{mSig}(\mathsf{C}_n) & \\
\mathit{mSig}(\mathsf{I}) = \mathit{mSig}(\overline{\mathsf{I}}) \cup \mathit{mSig}(\overline{\mathsf{S}}) & \textbf{if interface } \mathsf{I} \textbf{ extends } \overline{\mathsf{I}}\ \{\,\overline{\mathsf{S}};\,\} \\
\mathit{mSig}(\mathsf{I}_1,\dots,\mathsf{I}_n) = \mathit{mSig}(\mathsf{I}_1) \cup \dots \cup \mathit{mSig}(\mathsf{I}_n) & \\
\mathit{mSig}(\mathsf{I}\{\mathsf{R}\}) = \mathit{mSig}(\mathsf{I}) \cup \mathit{mSig}(\overline{\mathsf{S}}) \cup \mathit{mSig}(\overline{\mathsf{Z}}) \cup \mathit{mSig}(\overline{\mathsf{J}}) & \textbf{if replaceable } \mathsf{R} \textbf{ is } \{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}};\mid\overline{\mathsf{Z}};\mid\overline{\mathsf{J}};\rangle \\
\mathit{mSig}(\mathsf{U}_1,\dots,\mathsf{U}_n) = \mathit{mSig}(\mathsf{U}_1) \cup \dots \cup \mathit{mSig}(\mathsf{U}_n) &
\end{array}
$$

Figure 2: FDTJ: Lookup functions

signatures with the same method name); in the union of sequences, $\overline{\sigma} \cup \overline{\zeta}$, it is assumed that if $\mathsf{n} \in \mathit{names}(\overline{\sigma})$ and $\mathsf{n} \in \mathit{names}(\overline{\zeta})$ then $\mathit{choose}(\overline{\sigma},\mathsf{n}) = \mathit{choose}(\overline{\zeta},\mathsf{n})$; and in the disjoint union of sequences, $\overline{\sigma} \cdot \overline{\zeta}$, it is assumed that $\mathit{names}(\overline{\sigma}) \cap \mathit{names}(\overline{\zeta}) = \bullet$). We assume that all the source language types occurring in a program are well-formed.

4.2. Typing

In order to be useful in practice a JAVA-like nominal type system for static and dynamic traits has to support the typechecking of traits in isolation from both the trait (static) composition and (dynamic) replacement operations that use them, so that it is possible to typecheck a method defined in a trait only once (instead of having to typecheck it in every trait composition or replacement operation using that trait).

The FDTJ type system supports the above property through the use of constraints and the use of a suitable combination of nominal and structural typing. Within a basic trait expression, the uses of method parameters are type-checked according to the nominal notion of typing defined by the interface hierarchy and to a structural notion of typing for replaceables, while the uses of this are type-checked according to a structural notion of typing that takes into account the fields and methods *required* by the trait and the methods *provided* by the trait.

4.2.1. Types and Constraints

The syntax of source language types, ranged over by U and V, has been already given in Figure 1. *Pseudo-nominal types*, ranged over by π, are either class names or a source language types (remember that, as explained in Section 3.3, for every interface I we identify the types I{EmptyR} and I).[2] The syntax of pseudo-nominal types is as follows:

[2] The term "pseudo-nominal" aims to recall that these types include both nominal types and types of the shape I{R} (the source level types) that are composed by a nominal type (the interface name I) and by another component (the replaceable name R) that is not a nominal type.

$\boxed{\pi \ ::= \ \mathsf{C} \mid \mathsf{U}}$.

The syntax of the *structural types* for the this pseudo-variable, ranged over by τ, is as follows: $\boxed{\tau \ ::= \ \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \,\rangle}$. The pair $\langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \,\rangle$ specifies that this has the fields $\overline{\mathsf{F}}$ and methods with signatures $\overline{\sigma}$. The syntax of the *types for expressions*, ranged over by θ, is as follows: $\boxed{\theta \ ::= \ \pi \mid \tau}$. That is, a type for expressions is either a pseudo-nominal type or a structural type for this.

Besides assigning to each expression e a type describing the object yielded by the evaluation of e, the FDTJ type system infers also the constraints on this imposed by its use within e (*this-constraints*) and the constraints on traits imposed by the dynamic trait replacement expressions within e (*trait-constraints*).

- The syntax of this-constraints, ranged over by γ, is as follows: $\boxed{\gamma \ ::= \ \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \mid \overline{\mathsf{U}} \,\rangle}$. The triple $\langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \mid \overline{\mathsf{U}} \,\rangle$ specifies that the expression e selects the fields $\overline{\mathsf{F}}$ and the methods $\overline{\sigma}$ on this, and requires that this has the types $\overline{\mathsf{U}}$. In particular, the types $\overline{\mathsf{U}}$ are the types of the method's formal parameters to which this is passed inside the expression e. We recall that this will assume a meaning according to the class where the traits will be used.
- Trait-constraints, ranged over by Δ, are sets of trait-replaceable inclusions. The syntax of trait-replaceable inclusions, ranged over by δ, is as follows: $\boxed{\delta \ ::= \ \mathsf{T} \lessdot \mathsf{R}}$. The pair $\mathsf{T} \lessdot \mathsf{R}$ specifies that the set of methods defined by the trait T satisfies (according to Definition 4.6) the replaceable R.

Method types, ranged over by μ, are triples $\zeta \mid \gamma \mid \Delta$ where ζ is the method's signature and γ and Δ are the constraints inferred for the method's body. The typing rule for classes will check that the this-constraints inferred for the bodies of the methods of the class are satisfied. Trait-constraints will be checked after checking all the trait definitions in the program, when the typings of all traits mentioned in the constraints will be available (see Definition 4.16).

Remark 4.2. *The* FDTJ *type system collects constraints on a per-method basis, rather than on a per-trait basis (as usual in the nominal type systems that can be found in the literature). Collection constraints on a per-method basis has been proposed, in a structurally typed setting, by Reppy and Turon [36], and subsequently, in connection with a* JAVA*-like nominal type system, by Bono et al [9]. Both the proposals [36] and [9] do not consider dynamic trait replacement. Collecting method dependencies on a per-method basis is needed in order to be able to deal with the method exclusion operation (mentioned in Section 3.1). Since the* FDTJ *calculus does not include trait composition operations, the type system could be safely modified to collect constraints on a per-trait basis and to avoid to collect the constraints on fields and methods selected on* this *in the this-constraints inferred by the typing rules for expressions and method definitions given in Figure 5 (the typing rule for basic trait expressions, given in Figure 6, could simply take the declarations contained in the basic trait expressions). This would slightly simplify the presentation of the system. However, we have decided to collect constraint on a per-method basis since this makes it straightforward to extend the system to deal with trait composition operations by adapting the typing rules in [9].*

4.2.2. Subtyping Rules

To simplify the presentation of the subtyping rules we introduce the specificationof *operation for classes.* That is, a lookup function that, given a class name C, returns a replaceable declaration right-hand side that "characterizes" the set of methods of C.

Definition 4.3. *Let* **class** C **implements** $\overline{\mathsf{I}}$ **by** TE $\{\, \overline{\mathsf{F}};\ \mathsf{K} \,\}$. *We define* specificationof(C) *as* $\{mSig(\mathsf{C});\}\langle \overline{\mathsf{F}}; \mid \bullet \mid \overline{\mathsf{I}}; \rangle$.

Convention 4.4. *For every class* C*, we write* R_{C} *to denote a distinguished replaceable name (that cannot occur in source programs) and assume the replaceable definition* **replaceable** R_{C} **is** specificationof(C).

The subinterfacing relation, denoted by $\trianglelefteq$, is the reflexive and transitive closure of the immediate subinterfacing relation declared by the extends clauses in the interface table IT.

Replaceable inclusion and subtyping rules are given in Figure 3. The replaceable inclusion rule models the fact that, if the replaceable R is included into the replaceable R′, then every set of methods that satisfies (according to the explanation given in Section 3.3) R satisfies also R′. Note that the replaceable inclusion relation is reflexive and transitive. In order to ensure that replaceable declarations are independent from other replaceable declarations, we decided not to adopt a nominal inclusion relation between replaceables.

Replaceable inclusion rule:

$$\frac{\textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}};\shortmid\overline{\mathsf{Z}};\shortmid\overline{\mathsf{J}};\rangle \quad \textbf{replaceable}\ \mathsf{R}'\ \textbf{is}\ \{\overline{\mathsf{S}}';\}\langle\overline{\mathsf{G}}';\shortmid\overline{\mathsf{Z}}';\shortmid\overline{\mathsf{J}}';\rangle \atop \mathit{mSig}(\overline{\mathsf{S}})\subseteq\mathit{mSig}(\overline{\mathsf{S}}') \quad \overline{\mathsf{G}}\subseteq\overline{\mathsf{G}}' \quad \mathit{mSig}(\overline{\mathsf{S}}\cup\overline{\mathsf{Z}})\subseteq\mathit{mSig}(\overline{\mathsf{S}}'\cup\overline{\mathsf{Z}}') \quad \forall\mathsf{J}\in\overline{\mathsf{J}},\ \exists\mathsf{J}'\in\overline{\mathsf{J}}',\quad \mathsf{J}'\trianglelefteq\mathsf{J}}{\mathsf{R}\sqsubseteq\mathsf{R}'}$$

Subtyping rules:

$$\mathsf{C}<:\mathsf{C} \qquad \frac{\textbf{class}\ \mathsf{C}\ \textbf{implements}\ \overline{\mathsf{J}}\ \textbf{by}\cdots \quad \exists\mathsf{J}\in\overline{\mathsf{J}},\ \mathsf{J}\{\mathsf{R}_\mathsf{C}\}<:\mathsf{I}\{\mathsf{R}\}}{\mathsf{C}<:\mathsf{I}\{\mathsf{R}\}} \qquad \frac{\mathsf{I}\trianglelefteq\mathsf{I}' \quad \mathsf{R}\sqsubseteq\mathsf{R}'}{\mathsf{I}\{\mathsf{R}'\}<:\mathsf{I}'\{\mathsf{R}\}}$$

Figure 3: FDTJ: Replaceable inclusion and subtyping

The subtyping relation is syntax directed. The first rule is the standard reflexivity rule for classes. Recall the meaning of types of the form I{R} (illustrated in Section 3.3). The second rule exploits Definition 4.3 and Convention 4.4, and then relies on the third rule. The third rule exploits the subinterfacing and replaceable inclusion relations (replaceable inclusion is exploited contra-variantly because replaceables are requirements, so a subtype cannot have more requirements). Note that reflexivity and transitivity are admissible. Moreover, since the types I{EmptyR} and I are identified, subtyping is an extension of subinterfacing.

The rules integrate nominal subtyping (expressed by the implements clauses in the class table CT and by the subinterfacing relation) and structural subtyping (expressed by the replaceable inclusion relation, that exploits the replaceable component R of a source language type I{R} to ensure that the subtype has all the fields/methods and implements all the interfaces of the supertype).

Example 4.5. *Consider the interface* ISequence *in Listing 3, the interface* ISomeSequence *in Listing 9, the replaceable* RPolicy *in Listing 10, and the following replaceable*

```
replaceable RExtractionPolicy is { void out(); Object get(); } < List l; | | >
```

Both RExtractionPolicy $\sqsubseteq$ RPolicy *and* ISomeSequence{RPolicy} $<:$ ISequence{RExtractionPolicy} *hold.*

Consider also the replaceable RExample *in Listing 9 and the replaceable* RAnotherExample *in Example 4.7. Both* RAnotherExample $\sqsubseteq$ RExample *and* ISomeSequence{RExample} $<:$ ISomeSequence{RAnotherExample} *hold.*

4.2.3. On the Meaning of Replaceables and Replaceable Inclusion

A replaceable is a predicate over a set of methods (that is, over a trait), declared independently from any interface declaration and from other replaceable declarations. The following definition describes in a precise way, by relying on the subtyping relation defined in Figure 3, when a set of methods satisfies a replaceable (this definition has been already illustrated in Section 3.3, when no definition of the subtyping relation was available).

Definition 4.6. *A set of methods* satisfies *the replaceable* **replaceable** R **is** $\{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}};\shortmid\overline{\mathsf{Z}};\shortmid\overline{\mathsf{J}};\rangle$ *if and only if the set consists of methods whose signatures occur in mSig*$(\overline{\mathsf{S}})$ *and whose bodies*

- *select only the fields* $\overline{\mathsf{G}}$ *on* this*,*
- *select only the methods* $\overline{\mathsf{S}}\cdot\overline{\mathsf{Z}}$ *on* this*,*
- *assume only the interfaces* $\overline{\mathsf{J}}$ *as nominal types for* this*, that is,*
  - *pass* this *as argument to a method only if there exists* $\mathsf{J}\in\overline{\mathsf{J}}$ *such that* J{R} *is a subtype of the type of the corresponding formal parameter of the method, and*
  - *return* this *only if there exists* $\mathsf{J}\in\overline{\mathsf{J}}$ *such that* J{R} *is a subtype of the return type of the method.*

Example 4.7. *Both the trait* TFifoPolicy *in Listing 6 and* TinDisabled *in Listing 7 satisfy the replaceable* RPolicy *in Listing 10. Instead, the trait* TFifo *in Listing 3 does not satisfy the replaceable* RPolicy *(because of method* isEmpty*).*

Consider the replaceable RExample *and the traits* TExample1 *and* TExample2 *in Listing 9. Both* TExample1 *and* TExample2 *satisfy* RExample*. The fact that* TExample1 *satisfies* RExample *relies on the fact that* ISomeSequence{RExample} *is a subtype of* ISomeSequence *(according to the subtyping relation* $<:$ *in Figure 3). The trait* TExample2 *satisfies also the following replaceable*

Interface definition typing:

$$\frac{mSig(\mathsf{I}) = \cdots}{\vdash \textbf{interface}\ \mathsf{I}\ \textbf{extends}\ \overline{\mathsf{J}}\ \{\ \overline{\mathsf{S}}\ \}\ \ \mathrm{OK}} \qquad (\text{I-O}\textsc{k})$$

Replaceable definition typing:

$$\frac{names(\overline{\mathsf{S}}) \cap names(\overline{\mathsf{Z}}) = \bullet \qquad mSig(\overline{\mathsf{S}} \cup \overline{\mathsf{Z}}) \cup mSig(\overline{\mathsf{J}}) = \cdots}{\vdash \textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}}; \mid \overline{\mathsf{Z}}; \mid \overline{\mathsf{J}};\rangle\ \ \mathrm{OK}} \qquad (\text{R-O}\textsc{k})$$

Figure 4: FDTJ typing rules for interface definitions and replaceable definitions

```
replaceable RAnotherExample is { void outTwice(); boolean isNotEmpty(); boolean test(); }
  < ISomeSequenceComparator c; | void out(); | >
```

Instead, the trait TExample1 *does not satisfy* RAnotherExample *because the code of the method* isNotEmpty *selects the method* isEmpty *on* this *and the code of the method* test *selects the field* s *on* this *and passes* this *as argument to a method with formal parameter* ISomeSequence.

The following proposition illustrates the meaning of Definition 4.3 and Convention 4.4 in terms of Definition 4.6 (the proof is straightforward).

Proposition 4.8. *For every class* C*, if* $\overline{\mathsf{M}}$ *is the set of methods of* C*, then* $\overline{\mathsf{M}}$ *satisfies* R_{C}.

The following proposition states the soundness of the replaceable inclusion rule with respect to Definition 4.6 (the proof is straightforward).

Proposition 4.9. *If* $\mathsf{R} \sqsubseteq \mathsf{R}'$*, then every set of methods that satisfies* R *satisfies also* R'.

4.2.4. Typing Rules

A *type environment*, Γ, is either a finite mapping from variable names (including this) to types, written "$\overline{\mathsf{x}} : \overline{\mathsf{U}}, \mathsf{this} : \tau$", or the empty mapping, written "$\bullet$". The typing rules for interface definitions, replaceable definitions, expressions, method definitions, trait definitions and class definitions are syntax directed, with one rule for each form of term.

The typing judgment for **interface definitions** is $\boxed{\vdash \textbf{interface}\ \mathsf{I}\ \textbf{extends}\ \overline{\mathsf{I}}\ \{\ \overline{\mathsf{S}};\ \}\ \ \mathrm{OK}}$, to be read: "the definition of the interface I is well-typed". The associated rule, (I-Ok), is given in Figure 4. It exploits the lookup function $mSig(\cdot)$ to check that there are no conflicts in the signatures of the methods declared in the interface and in all its superinterfaces.

The typing judgment for **replaceable definitions** is $\boxed{\vdash \textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}}; \mid \overline{\mathsf{Z}}; \mid \overline{\mathsf{J}};\rangle\ \ \mathrm{OK}}$, to be read: "the definition of the replaceable R is well-typed". The associated rule, (R-Ok), is given in Figure 4. It checks that, within a replaceable definition, the signatures of the *methods that may be replaced* ($\overline{\mathsf{S}}$) and the signatures of the *methods that may be used by the replaced methods but may not be replaced* ($\overline{\mathsf{Z}}$) are disjoint, and that there are no conflicts in all the the signatures of the methods declared in the trait ($\overline{\mathsf{S}} \cup \overline{\mathsf{Z}}$) or belonging to the *interfaces that may be used as types for* this ($\overline{\mathsf{J}}$).

The typing judgment for **expressions** is $\boxed{\Gamma \vdash \mathsf{e} : \theta \mid \gamma \mid \Delta}$ to be read: "under the assumption in Γ, the expression e is well-typed with type θ and constraints γ and Δ". The associated rules are given in Figure 5. The type of the pseudo-variable this is the structural type $\Gamma(\mathsf{this})$. The type of an object creation expression **new** C(...) is the class name C. The type of any other FDTJ expression e is a source language type, i.e., a type of the form $\mathsf{I}\{\mathsf{R}\}$ for some interface name I and replaceable name R. The most interesting rules are the following.

- Rule (T-InvkThis) checks method invocation when the receiver is the this pseudo-variable. First the signature of m ($\mathsf{U}\,\mathsf{m}\,(\mathsf{U}_1, ..., \mathsf{U}_n) = \zeta$) is extracted from the sequence $\overline{\sigma}$ associated to this. Then, the types of the actual parameters $\mathsf{e}_1, ..., \mathsf{e}_n$ are checked. For the arguments that are different from this (e_i such that $i \notin \mathscr{T}$) the subtyping check between actual and formal parameter types is performed ($\theta_i <: \mathsf{U}_i$). This check is not performed when

Expression typing:

$$\Gamma \vdash \mathsf{x} : \Gamma(\mathsf{x}) \mid \langle \bullet \mid \bullet \mid \bullet \rangle \mid \emptyset \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathsf{this} : \langle\, \overline{\mathsf{F}} \mid ... \,\rangle \mid \langle \bullet \mid \bullet \mid \bullet \rangle \mid \emptyset \qquad \mathit{choose}(\overline{\mathsf{F}},\mathsf{f}) = \mathsf{U\,f}}{\Gamma \vdash \mathsf{this.f} : \mathsf{U} \mid \langle\, \mathsf{U\,f} \mid \bullet \mid \bullet \,\rangle \mid \emptyset} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{l} \Gamma \vdash \mathsf{this} : \langle\, ... \mid \overline{\sigma} \,\rangle \mid \langle \bullet \mid \bullet \mid \bullet \rangle \mid \emptyset \qquad \forall i \in 1..n, \quad \Gamma \vdash \mathsf{e}_i : \theta_i \mid \langle\, \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{V}}^{(i)} \,\rangle \mid \Delta^{(i)} \\ \mathsf{U\,m}\,(\mathsf{U}_1,...,\mathsf{U}_n) = \mathit{choose}(\overline{\sigma},\mathsf{m}) = \zeta \\ \mathscr{T} = \{i \mid i \in 1..n \textbf{ and } \theta_i = \Gamma(\mathsf{this})\} \qquad \forall i \in 1..n - \mathscr{T}, \quad \theta_i <: \mathsf{U}_i \end{array}}{\Gamma \vdash \mathsf{this.m}(\mathsf{e}_1,...,\mathsf{e}_n) : \mathsf{U} \mid \langle\, \cup_{i\in 1..n}\overline{\mathsf{F}}^{(i)} \mid \zeta \cup (\cup_{i\in 1..n}\overline{\sigma}^{(i)}) \mid (\cup_{i\in 1..n}\overline{\mathsf{V}}^{(i)}) \cup (\cup_{i\in\mathscr{T}}\mathsf{U}_i) \,\rangle \mid \cup_{i\in 1..n}\Delta^{(i)}} \qquad \text{(T-INVKTHIS)}$$

$$\frac{\begin{array}{l} \mathsf{e}_0 \neq \mathsf{this} \qquad \forall i \in 0..n, \quad \Gamma \vdash \mathsf{e}_i : \theta_i \mid \langle\, \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{V}}^{(i)} \,\rangle \mid \Delta^{(i)} \\ \mathsf{U\,m}\,(\mathsf{U}_1,...,\mathsf{U}_n) = \mathit{choose}(\mathit{mSig}(\mathit{interfaces}(\theta_0)),\mathsf{m}) \\ \mathscr{T} = \{i \mid i \in 1..n \textbf{ and } \theta_i = \Gamma(\mathsf{this})\} \qquad \forall i \in 1..n - \mathscr{T}, \quad \theta_i <: \mathsf{U}_i \end{array}}{\Gamma \vdash \mathsf{e}_0.\mathsf{m}(\mathsf{e}_1,...,\mathsf{e}_n) : \mathsf{U} \mid \langle\, \cup_{i\in 0..n}\overline{\mathsf{F}}^{(i)} \mid \cup_{i\in 0..n}\overline{\sigma}^{(i)} \mid (\cup_{i\in 0..n}\overline{\mathsf{V}}^{(i)}) \cup (\cup_{i\in\mathscr{T}}\mathsf{U}_i) \,\rangle \mid \cup_{i\in 0..n}\Delta^{(i)}} \qquad \text{(T-INVKNONTHIS)}$$

$$\frac{\begin{array}{l} \mathit{fields}(\mathsf{C}) = \mathsf{U}_1\,\mathsf{f}_1;...;\mathsf{U}_n\,\mathsf{f}_n; \qquad \forall i \in 1..n, \quad \Gamma \vdash \mathsf{e}_i : \theta_i \mid \langle\, \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{V}}^{(i)} \,\rangle \mid \Delta^{(i)} \\ \mathscr{T} = \{i \mid i \in 1..n \textbf{ and } \theta_i = \Gamma(\mathsf{this}) = \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \,\rangle\} \qquad \forall i \in 1..n - \mathscr{T}, \quad \theta_i <: \mathsf{U}_i \end{array}}{\Gamma \vdash \textbf{new}\ \mathsf{C}(\mathsf{e}_1,...,\mathsf{e}_n) : \mathsf{C} \mid \langle\, \cup_{i\in 1..n}\overline{\mathsf{F}}^{(i)} \mid \cup_{i\in 1..n}\overline{\sigma}^{(i)} \mid (\cup_{i\in 1..n}\overline{\mathsf{V}}^{(i)}) \cup (\cup_{i\in\mathscr{T}}\mathsf{U}_i) \,\rangle \mid \cup_{i\in 1..n}\Delta^{(i)}} \qquad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash \mathsf{e} : \mathsf{I}\{\mathsf{R}\} \mid \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \mid \overline{\mathsf{U}} \,\rangle \mid \Delta \qquad \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \ldots}{\Gamma \vdash \mathsf{e}\{\mathsf{T}\} : \mathsf{I}\{\mathsf{R}\} \mid \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \mid \overline{\mathsf{U}} \,\rangle \mid \Delta \cup \{\mathsf{T} \lessdot \mathsf{R}\}} \qquad \text{(T-REPL)}$$

Method definition typing:

$$\frac{\begin{array}{l} \mathsf{this} : \tau, \overline{\mathsf{x}} : \overline{\mathsf{V}} \vdash \mathsf{e} : \theta \mid \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \mid \overline{\mathsf{U}} \,\rangle \mid \Delta \\ \theta = \tau \ \textbf{implies}\ \overline{\mathsf{U}}' = \overline{\mathsf{U}} \cup \mathsf{V} \\ \theta \neq \tau \ \textbf{implies}\ (\ \theta <: \mathsf{V}\ \textbf{and}\ \overline{\mathsf{U}}' = \overline{\mathsf{U}}\ ) \end{array}}{\mathsf{this} : \tau \vdash \mathsf{V\ m}\ (\overline{\mathsf{V}}\,\overline{\mathsf{x}})\{\textbf{return}\ \mathsf{e};\} : \mathsf{V\ m}\ (\overline{\mathsf{V}}) \mid \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \mid \overline{\mathsf{U}}' \,\rangle \mid \Delta} \qquad \text{(M-OK)}$$

Figure 5: FDTJ typing rules for expressions and method definitions

the actual parameter is this (e_i such that $i \in \mathscr{T}$) since no assumption about the nominal types of this can be made; instead the source language types used as types for this (U_i for $i \in \mathscr{T}$) are included in the this-constraints collected in the conclusion of the rule. Then, the this-constraints and the trait-constraints imposed by the expressions involved in method invocation (the actual parameters e_i) are collected in the conclusion. Also the constraint that this must have a method m with signature ζ is collected.

- Rule (T-INVKNONTHIS) checks method invocation when the receiver is not this is similar. First the signature of m ($\mathsf{U\,m}\,(\mathsf{U}_1,...,\mathsf{U}_n)$) is extracted from by the type θ_0 of the receiver e_0. Then, the types of the actual parameters $\mathsf{e}_1,...,\mathsf{e}_n$ are checked. and the this-constraints and the trait-constraints imposed by the expressions involved in method invocation (the receiver e_0 and the actual parameters $\mathsf{e}_1,...\mathsf{e}_n$) are collected in the conclusion.
- Rule (T-NEW), for instance creation, is similar (but it uses the fields declared in the class to check the arguments of the constructor).
- Rule (T-REPL) checks the dynamic replacement $\mathsf{e}\{\mathsf{T}\}$. In the conclusion the trait-constraints are updated with $\{\mathsf{T} \lessdot \mathsf{R}\}$ since the trait T is used for a dynamic replacement on e of type $\mathsf{I}\{\mathsf{R}\}$.

Example 4.10. *Consider the interface* ISequence *in Listing 3, the trait* TFifoPolicy *in Listing 6, the replaceable* RPolicy *in Listing 10, and the trait replacement expression* s{TFifoPolicy}. *We have:*

$$\begin{array}{l} \mathsf{this} : ..., \mathsf{s{:}ISequence}\{\mathsf{RPolicy}\} \vdash \mathsf{s}\{\mathsf{TFifoPolicy}\} : \\ \quad \mathsf{ISequence}\{\mathsf{RPolicy}\} \mid \langle \bullet \mid \bullet \mid \bullet \rangle \mid \{\mathsf{TFifoPolicy} \lessdot \mathsf{RPolicy}\} \end{array}$$

Trait expression typing:

$$\frac{\begin{array}{l} \mathit{mSig}(\overline{\mathsf{S}}) = \overline{\sigma} \qquad \mathit{mSig}(\mathsf{M}_1...\mathsf{M}_p) = \zeta_1...\zeta_p \qquad p \geq 0 \\ \forall i \in 1..p, \qquad \mathsf{this} : \langle\, \overline{\mathsf{F}} \mid \overline{\sigma} \cdot \zeta_1...\zeta_p \,\rangle \vdash \mathsf{M}_i : \mu_i \qquad \mu_i = \zeta_i \mid \langle\, \overline{\mathsf{F}}^{(i)} \mid \overline{\zeta}^{(i)} \mid \overline{\mathsf{U}}^{(i)} \,\rangle \mid \Delta^{(i)} \\ \overline{\mathsf{F}} = \overline{\mathsf{F}}^{(1)} \cup \cdots \cup \overline{\mathsf{F}}^{(p)} \qquad \overline{\sigma} = \mathit{exclude}((\overline{\zeta}^{(1)} \cup \cdots \cup \overline{\zeta}^{(p)}), \mathit{names}(\zeta_1...\zeta_p)) \\ \overline{\mathsf{F}} \cup \mathit{fields}(\overline{\mathsf{U}}^{(1)}) \cup \cdots \cup \mathit{fields}(\overline{\mathsf{U}}^{(p)}) = \cdots \qquad \zeta_1...\zeta_p \cup \overline{\zeta}^{(1)} \cup \cdots \cup \overline{\zeta}^{(p)} \cup \mathit{mSig}(\overline{\mathsf{U}}^{(1)}) \cup \cdots \cup \mathit{mSig}(\overline{\mathsf{U}}^{(p)}) = \cdots \end{array}}{\vdash \{\, \overline{\mathsf{F}};\ \overline{\mathsf{S}};\ \mathsf{M}_1...\mathsf{M}_p \,\} : \mu_1...\mu_p} \qquad \text{(T-TEBASIC)}$$

$$\frac{\vdash \textbf{trait}\ \mathsf{T} \cdots : \overline{\mu}}{\vdash \mathsf{T} : \overline{\mu}} \qquad \text{(T-TENAME)}$$

Trait definition typing:

$$\frac{\vdash \mathsf{TE} : \overline{\mu}}{\vdash \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \mathsf{TE} : \overline{\mu}} \qquad \text{(T-OK)}$$

Class definition tying:

$$\frac{\begin{array}{l} \vdash \mathsf{TE} : \mu_1...\mu_p \qquad p \geq 0 \qquad \forall i \in 1..p, \quad \mu_i = \zeta_i \mid \langle\, \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{U}}^{(i)} \,\rangle \mid \ldots \\ \overline{\mathsf{V}}\,\overline{\mathsf{g}} \supseteq \overline{\mathsf{F}}^{(1)} \cdots \cup \overline{\mathsf{F}}^{(p)} \cup \mathit{fields}(\overline{\mathsf{U}}^{(1)}) \cup \cdots \cup \mathit{fields}(\overline{\mathsf{U}}^{(p)}) \\ \zeta_1...\zeta_p \supseteq \overline{\sigma}^{(1)} \cup \cdots \cup \overline{\sigma}^{(p)} \cup \mathit{mSig}(\overline{\mathsf{U}}^{(1)}) \cup \cdots \cup \mathit{mSig}(\overline{\mathsf{U}}^{(p)}) \cup \mathit{mSig}(\overline{\mathsf{I}}) \\ \forall \mathsf{I}' \in \cup_{i \in 1..p} \mathit{allInterfaces}(\overline{\mathsf{U}}^{(i)}), \quad \exists \mathsf{I} \in \overline{\mathsf{I}}, \quad \mathsf{I} \trianglelefteq \mathsf{I}' \end{array}}{\vdash\ \textbf{class}\ \mathsf{C}\ \textbf{implements}\ \overline{\mathsf{I}}\ \textbf{by}\ \mathsf{TE}\ \{\ \overline{\mathsf{V}}\,\overline{\mathsf{g}};\ \ \mathsf{C}(\overline{\mathsf{V}}\,\overline{\mathsf{g}})\ \{\ \mathsf{this}.\overline{\mathsf{g}} = \overline{\mathsf{g}};\ \}\ \}\ \ \mathrm{OK}} \qquad \text{(C-OK)}$$

Figure 6: FDTJ typing rules for trait expressions, trait definitions and class definitions

Consider the interface ISomeSequence *and the trait* TExample2 *in Listing 9, the replaceable* RAnotherExample *in Example 4.7, and the trait replacement expression* x{TExample2}*. We have:*

$$\begin{array}{l} \mathsf{this} : ..., \mathsf{x{:}ISomeSequence\{RAnotherExample\}} \vdash \mathsf{x\{TExample2\}} : \\ \qquad \mathsf{ISomeSequence\{RAnotherExample\}} \mid \langle\, \bullet \mid \bullet \mid \bullet \,\rangle \mid \{\mathsf{TExample2} \lessdot \mathsf{RAnotherExample}\} \end{array}$$

The typing rule judgement for **method definitions** is $\boxed{\mathsf{this} : \tau \vdash \mathsf{V}\ \mathsf{m}\ (\overline{\mathsf{V}}\ \overline{\mathsf{x}})\{\textbf{return}\ \mathsf{e};\} : \mu}$, where $\mu = \zeta \mid \gamma \mid \Delta$. To be read: "under the assumption that this has type τ, the definition of method m is well-typed with type μ". That is, the method m has signature ζ ($= \mathsf{V}\ \mathsf{m}\ (\overline{\mathsf{V}})$) and its body is type correct if the enclosing class satisfies the constraints γ and Δ. The associated rule, (M-OK), is given in Figure 5. There are two cases:

- If the type of the method body is τ (i.e., the method simply returns this), then the sequence of source language types required for this is updated adding V (i.e., the return type of the method).
- Otherwise, the type of the body must be a subtype of the return type of the method.

The typing judgement for **trait expressions** is $\boxed{\vdash \mathsf{TE} : \overline{\mu}}$ where $\overline{\mu} = \mu_1...\mu_n$ $(n \geq 0)$. To be read: "the trait expression TE is well-typed with type $\overline{\mu}$". That is, TE provides n methods with types $\mu_1,...,\mu_n$, respectively. The associated rules, (T-TEBASIC) and (T-TENAME), are given in Figure 6.

- Rule (T-TEBASIC) checks that each method M_i defined by the trait has a type μ_i. The check is performed by assuming for this a type consisting of all the fields required by the trait ($\overline{\mathsf{F}}$) and of all the signatures of the methods required by the trait ($\overline{\sigma} = \mathit{mSig}(\overline{\mathsf{S}})$) and all the signatures of the methods defined by the trait ($\zeta_1...\zeta_p$). Then, it checks that all the methods together requires all the field requirements declared in the trait ($\overline{\mathsf{F}} = \overline{\mathsf{F}}^{(1)} \cup \cdots \cup \overline{\mathsf{F}}^{(p)}$), and that all the methods together requires all the method requirements declared in the trait ($\overline{\sigma} = \mathit{exclude}((\overline{\zeta}^{(1)} \cup \cdots \cup \overline{\zeta}^{(p)}), \mathit{names}(\zeta_1...\zeta_p))$). Finally, it checks that there are no conflicts among the fields/signatures declarations of the trait and the this-constraints inferred for the methods.

- Rule (T-TENAME) just assigns to the trait name T the type inferred for the definition of the trait T.

The typing judgement for **trait definitions** is $\boxed{\vdash \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \mathsf{TE} : \overline{\mu}}$, to be read: "the definition of trait T is well-typed with type $\overline{\mu}$". The associated rule, (T-OK), is given in Figure 6. It assigns to the the trait T the type inferred for the trait expression TE (remember that the trait reuse graph is acyclic).

Example 4.11. *Consider the trait* TExample1 *in Listing 9. Let*

$$\overline{\mathsf{F}} \;=\; \mathsf{ISomeSequence\ s;\ ISomeSequenceComparator\ c;}$$

be the required fields; let σ_{out} *and* $\sigma_{\mathsf{isEmpty}}$ *be the signatures of the required methods; let* $\sigma_{\mathsf{outTwice}}$, $\sigma_{\mathsf{isNotEmpty}}$, σ_{test} *and* $\mathsf{M}_{\mathsf{outTwice}}$, $\mathsf{M}_{\mathsf{isNotEmpty}}$ *and* $\mathsf{M}_{\mathsf{test}}$ *be the signatures and the definitions of the provided methods, respectively; and let*

$$\overline{\sigma} \;=\; \sigma_{\mathsf{out}}\,\sigma_{\mathsf{isEmpty}}\,\sigma_{\mathsf{outTwice}}\,\sigma_{\mathsf{isNotEmpty}}\,\sigma_{\mathsf{test}}$$

We have:

$$\begin{array}{l}
\mathsf{this} : \langle\, \overline{\mathsf{F}} \mid \overline{\sigma}\,\rangle \vdash \mathsf{M}_{\mathsf{outTwice}} : \mu_{\mathsf{outTwice}}\\
\mathsf{this} : \langle\, \overline{\mathsf{F}} \mid \overline{\sigma}\,\rangle \vdash \mathsf{M}_{\mathsf{isNotEmpty}} : \mu_{\mathsf{isNotEmpty}}\\
\mathsf{this} : \langle\, \overline{\mathsf{F}} \mid \overline{\sigma}\,\rangle \vdash \mathsf{M}_{\mathsf{test}} : \mu_{\mathsf{test}}\\
\vdash \mathsf{TExample} : \mu_{\mathsf{outTwice}}\,\mu_{\mathsf{isNotEmpty}}\,\mu_{\mathsf{test}}
\end{array}$$

where

$$\begin{array}{lcl}
\mu_{\mathsf{outTwice}} & = & \sigma_{\mathsf{outTwice}} \mid \langle\, \bullet \mid \mathsf{void\ out()} \mid \bullet\,\rangle \mid \emptyset\\
\mu_{\mathsf{isNotEmpty}} & = & \sigma_{\mathsf{isNotEmpty}} \mid \langle\, \bullet \mid \mathsf{boolean\ isEmpty()} \mid \bullet\,\rangle \mid \emptyset\\
\mu_{\mathsf{test}} & = & \sigma_{\mathsf{test}} \mid \langle\, \overline{\mathsf{F}} \mid \bullet \mid \mathsf{ISomeSequence}\,\rangle \mid \emptyset
\end{array}$$

The typing judgement for **class definitions** is $\boxed{\vdash \textbf{class}\ \mathsf{C}\ \textbf{implements}\ \bar{\mathsf{I}}\ \textbf{by}\ \mathsf{TE}\ \{\ \overline{\mathsf{F}};\ \mathsf{K}\ \}\quad \mathrm{OK}}$, to be read: "the definition of the class C is well-typed". The associated typing rule, (C-OK), is given in Figure 6. It checks that the this-constraints inferred for the methods provided by the trait expression TE are satisfied. That is: that the class provides all the fields expected by the trait; that the trait provides all the methods that the class needs to correctly implement the interfaces $\bar{\mathsf{I}}$; and that, for each interface in the this-constraints of the trait, there is a subinterface implemented by the class.

4.2.5. Trait-Constraint Checking and Well-Typed Programs

In order to simplify the presentation of the trait-constraints checking rule and of the notion of well typed program we introduce the specificationof operation and the traitconstraintsof operation. Given a trait name T: specificationof(T) returns a replaceable definition right-hand side that "characterizes" the set of methods defined by T, and and traitconstraintsof(T) returns the trait-constraints for the methods defined by T. In the following definition we will use two lookup functions: *fields* (that given a source language type I{R} returns the sequence of the fields required by the replaceable R) and *allInterfaces* (that given I{R} returns the sequence formed by the interface name I and all the interfaces names listed in R) defined in Section 4.1.1.

Definition 4.12. *Let* $\vdash \mathsf{T} : \mu_1 \ldots \mu_p$ *where (for all* $i \in 1..p$*)* $\mu_i = \zeta_i \mid \langle\, \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{U}}^{(i)}\,\rangle \mid \Delta^{(i)}$.

1. *We define* specificationof(T) *as* $\{\overline{\zeta};\}\langle\overline{\mathsf{F}}; \mid \overline{\sigma}; \mid \bar{\mathsf{I}};\rangle$, *where* $\overline{\zeta} = \zeta_1 \ldots \zeta_p$, $\overline{\mathsf{F}} = (\cup_{i\in 1..p}\overline{\mathsf{F}}^{(i)}) \cup (\cup_{i\in 1..p}\mathit{fields}(\overline{\mathsf{U}}^{(i)}))$, $\overline{\sigma} = \overline{\zeta} \cup (\cup_{i\in 1..p}\overline{\sigma}^{(i)}) \cup (\cup_{i\in 1..p}\mathit{mSig}(\overline{\mathsf{U}}^{(i)}))$, *and* $\bar{\mathsf{I}} = \mathit{allInterfaces}(\cup_{i\in 1..p}\overline{\mathsf{U}}^{(i)})$.
2. *We define* traitconstraintsof(T) *as* $\cup_{i\in 1..p}\Delta^{(i)}$.

Observe that, while the specificationof operation for classes (in Definition 4.3) does not rely on the typing rules, the value of specificationof(T) and traitconstraintsof(T) can be computed only when the typing of the trait T is available.

Example 4.13. *Consider the replaceable* RExample *and the trait* TExample1 *in Listing 9. We have that* specificationof(TExample1) *yields the left-hand side of the definition of the replaceable* RExample.

$$\frac{\vdash \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \mathsf{TE} : \mu_1 ... \mu_p \qquad \textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\overline{\mathsf{S}};\}\langle\overline{\mathsf{G}}; \mid \overline{\mathsf{Z}}; \mid \overline{\mathsf{J}};\rangle \qquad \mathsf{R}_\mathsf{T} \sqsubseteq \mathsf{R}}{\vdash \mathsf{T} \lessdot \mathsf{R}\ \ \mathrm{OK}} \quad (\lessdot\text{-O}\textsc{k})$$

Figure 7: FDTJ checking rule for trait-constraints

Convention 4.14. *For every trait* T*, we write* R_T *to denote a distinguished replaceable name (that cannot occur in source programs) and assume the replaceable definition* $\textbf{replaceable}\ \mathsf{R}_\mathsf{T}\ \textbf{is}\ \mathsf{specificationof}(\mathsf{T})$.

The following proposition illustrates the meaning of Definition 4.12 and Convention 4.14 in terms of Definition 4.6 (the proof is straightforward).

Proposition 4.15. *For every trait* T*, if* $\overline{\mathsf{M}}$ *is the set of methods of* T*, then* $\overline{\mathsf{M}}$ *satisfies* R_T.

The judgement for trait-constraint checking is $\boxed{\vdash \mathsf{T} \lessdot \mathsf{R}\ \ \mathrm{OK}}$ to be read: "the constraint $\mathsf{T} \lessdot \mathsf{R}$ is satisfied". The associated typing rule, ($\lessdot$-OK), is given in Figure 7. The rule exploits Definition 4.12 and Convention 4.14. It checks that the trait T can actually replace the methods specified in the replaceable R by relying on the replaceable inclusion relation of Figure 3.

Definition 4.16 (Well-typed FDTJ programs). *We write* $\vdash_{\mathrm{FDTJ}} (\mathsf{IT},\mathsf{TT},\mathsf{RT},\mathsf{CT},\mathsf{e}) : \pi$*, to be read: "the program* $(\mathsf{IT},\mathsf{TT},\mathsf{RT},\mathsf{CT},\mathsf{e})$ *is well-typed with type* π*", to mean that*

- *the interfaces in* IT*, the replaceables in* RT*, the traits in* TT *and the classes in* CT *are well-typed,*
- *all the trait-constraints for the methods defined by the traits in* TT *are satisfied (that is, for all trait* T *in* TT *and for all* δ *in* $\mathsf{traitconstraintsof}(\mathsf{T})$ *the judgement* $\vdash \delta\ \ \mathrm{OK}$ *holds), and*
- *the expression* e *is well typed with type* π*, empty this-constraints and some satisfied trait-constraints* Δ*, under the empty set of assumptions (that is, the judgment* $\bullet \vdash \mathsf{e} : \pi \mid \langle\ \bullet \mid \bullet \mid \bullet\ \rangle \mid \Delta$ *holds and, for all* $\delta \in \Delta$*, the judgment* $\vdash \delta\ \ \mathrm{OK}$ *holds).*

Note that the expression to be executed, e, is *closed* (that is, it does not contain variables).

4.3. Reduction

Dynamic trait replacement is an imperative operation working on a per-object basis. That is, while the methods associated to the object's class remain unchanged, the object must keep track of the methods that have been introduced by means of dynamic replacements.

In order to model object identities and dynamic replacements we need to model the notions of address, object and heap. *Addresses*, ranged over by the metavariable ι, are the elements of the denumerable set $\mathbf{I}$. An *object* is a triple $\langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota},\ \overline{\mathsf{M}}\rangle$, where C is the object's class, $\overline{\mathsf{f}} : \overline{\iota}$ is a mapping from the names of the object's fields to their *values* (i.e., addresses) and $\overline{\mathsf{M}}$ are the object's methods that have been introduced by means of dynamic replacements (therefore, immediately after object creation this set will be empty). A *heap*, $\mathscr{H}$, is a mapping from addresses to objects. The empty heap will be denoted by $\emptyset$.

The states of a computation are represented by means of configurations. A *configuration* is a pair "$e,\mathscr{H}$", where e is a *runtime expression* (i.e., the code under evaluation) and $\mathscr{H}$ is a heap. Runtime expressions, ranged over by e, are obtained from source language expressions by removing variables, adding addresses ι (i.e., values) and replacing field selections $\mathsf{this.f}$ by $\iota.\mathsf{f}$.

The reduction relation has the form $\boxed{e,\mathscr{H} \longrightarrow e',\mathscr{H}'}$, read "configuration $e,\mathscr{H}$ reduces to configuration $e',\mathscr{H}'$ in one step". We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. The reduction rules, given in Figure 8, ensure that the computation is carried on according to a call-by-value reduction strategy by using the standard notions of computation rules and congruence rules.

The most interesting rules are the computation rules. The rule for field selection, (R-FIELD), returns the value associated to the selected field. The rule for object creation, (R-NEW), stores the newly created object at a fresh address

Computation rules:

$$\frac{\mathcal{H}(\iota) = \langle \mathsf{C},\ \ldots, \mathsf{f}_i : \iota_i, \ldots,\ \overline{\mathsf{M}} \rangle}{\iota.\mathsf{f}_i, \mathcal{H} \longrightarrow \iota_i, \mathcal{H}} \qquad \text{(R-FIELD)}$$

$$\frac{\mathcal{H}(\iota) = \langle \mathsf{C},\ \bar{\mathsf{f}} : \bar{\iota},\ \overline{\mathsf{M}} \rangle \qquad \mathsf{U}\ \mathsf{m}\ (\overline{\mathsf{U}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \mathit{exclude}(\mathit{methods}(\mathsf{C}), \mathit{names}(\overline{\mathsf{M}})) \cup \overline{\mathsf{M}}}{\iota.\mathsf{m}(\bar{\iota}), \mathcal{H} \longrightarrow e[{}^{\iota}/\mathsf{this}, {}^{\bar{\iota}}/\overline{\mathsf{x}}], \mathcal{H}} \qquad \text{(R-INVK)}$$

$$\frac{\iota \notin \mathrm{Dom}(\mathcal{H}) \qquad \mathit{fields}(\mathsf{C}) = \overline{\mathsf{F}}\ \bar{\mathsf{f}}}{\textbf{new}\ \mathsf{C}(\bar{\iota}), \mathcal{H} \longrightarrow \iota, \mathcal{H} \cup \{\iota \mapsto \langle \mathsf{C},\ \bar{\mathsf{f}} : \bar{\iota},\ \bullet \rangle\}} \qquad \text{(R-NEW)}$$

$$\frac{\mathcal{H}(\iota) = \langle \mathsf{C},\ \bar{\mathsf{f}} : \bar{\iota},\ \overline{\mathsf{M}} \rangle \qquad \mathit{methods}(\mathsf{T}) = \overline{\mathsf{M}}'}{\iota\{\mathsf{T}\}, \mathcal{H} \longrightarrow \iota, \mathcal{H}[\iota \mapsto \langle \mathsf{C},\ \bar{\mathsf{f}} : \bar{\iota},\ \mathit{exclude}(\overline{\mathsf{M}}, \mathit{names}(\overline{\mathsf{M}}')) \cup \overline{\mathsf{M}}' \rangle]} \qquad \text{(R-REPL)}$$

Congruence rules:

$$\frac{e, \mathcal{H} \longrightarrow e', \mathcal{H}'}{e.\mathsf{f}, \mathcal{H} \longrightarrow e'.\mathsf{f}, \mathcal{H}'} \qquad \frac{e, \mathcal{H} \longrightarrow e', \mathcal{H}}{e\{\mathsf{T}\}, \mathcal{H} \longrightarrow e'\{\mathsf{T}\}, \mathcal{H}} \qquad \frac{e, \mathcal{H} \longrightarrow e', \mathcal{H}'}{e.\mathsf{m}(\bar{e}), \mathcal{H} \longrightarrow e'.\mathsf{m}(\bar{e}), \mathcal{H}'}$$

$$\frac{e_i, \mathcal{H} \longrightarrow e'_i, \mathcal{H}'}{\iota_0.\mathsf{m}(\bar{\iota}, e_i, \bar{e}), \mathcal{H} \longrightarrow \iota_0.\mathsf{m}(\bar{\iota}, e'_i, \bar{e}), \mathcal{H}'} \qquad \frac{e_i, \mathcal{H} \longrightarrow e'_i, \mathcal{H}'}{\textbf{new}\ \mathsf{C}(\bar{\iota}, e_i, \bar{e}), \mathcal{H} \longrightarrow \textbf{new}\ \mathsf{C}(\bar{\iota}, e'_i, \bar{e}), \mathcal{H}'}$$

Figure 8: FDTJ reduction rules

of the heap and returns the address. Object's fields are initialized as specified by the class constructor. The rule for method invocation, (R-INVK), searches for the method definition for m in the set of replaced methods $\overline{\mathsf{M}}$, and if it cannot find it then it relies on the method definition that is found in the original object ($\mathit{exclude}(\mathit{methods}(\mathsf{C}), \mathit{names}(\overline{\mathsf{M}})) \cup \overline{\mathsf{M}}$). Note that this lookup will always succeed in a well-typed program. Rule (R-REPL) performs method replacement; since replacements can be performed many times on an object, we must take care of removing methods that were previously replaced.

5. Properties

The type soundness result for the FDTJ calculus is as follows.

Theorem 5.1 (FDTJ **Type Soundness).** *Let* $\bullet \vdash \mathsf{e}_0 : \pi \mid \langle \bullet \mid \bullet \mid \bullet \rangle \mid \Delta$ *and* $\vdash \delta\ \ \mathrm{OK}$ *for all* $\delta \in \Delta$*. If* $\mathsf{e}_0, \emptyset \rightarrow^{\star} e, \mathcal{H}$ *with e a normal form, then the heap* $\mathcal{H}$ *is well formed and e is an address* ι *such that* $\mathcal{H}(\iota) = \langle \mathsf{C}, \cdots, \cdots \rangle$ *and* $\mathsf{C} <: \pi$.

To prove the type soundness result we will introduce a suitable notion of typing for configurations (that is, runtime expressions and heaps). The objects in the heap may contain methods introduced by means of dynamic replacement, therefore typing a configuration may involve typing method bodies (that is, source language expressions). We address this issue by introducing the notion of *open runtime expressions*, that is, runtime expressions containing variables (which encompass both source language expressions and runtime expressions). The notion of open runtime expressions allowed us to simplify the structure of the proof of the properties that relate source language typing with runtime typing. In particular, it makes it possible to relate the source language typing of a method method body e (which may contain variables) to the runtime typing of e and then to rely on the substitution lemma for open runtime expression typing (Lemma A.3 in the appendix).

A *runtime type environment* Σ is a finite (possible empty) mapping that: either *(i)* maps addresses to class names; or *(ii)* maps this to a class name and maps variable names (different from this) to source language types (that are types of the form $\mathsf{I}\{\mathsf{R}\}$).

The typing judgement for (runtime or source language) expressions is $\boxed{\Sigma \vdash' e : \pi}$ to be read: "under the assumptions in Σ, the (runtime or source language) expression e is well-typed with type π". Note that the conclusion of the

Open runtime expression typing:

$$\Sigma \vdash' \mathsf{x} : \Sigma(\mathsf{x}) \qquad \text{(RT-VAR)}$$

$$\Sigma \vdash' \iota : \Sigma(\iota) \qquad \text{(RT-ADDR)}$$

$$\frac{\Sigma \vdash' e : \mathsf{C} \qquad \mathit{fields}(\mathsf{C}) = \overline{\mathsf{F}} \qquad \mathsf{U\,f} \in \overline{\mathsf{F}}}{\Sigma \vdash' e.\mathsf{f} : \mathsf{U}} \qquad \text{(RT-FIELD)}$$

$$\frac{\Sigma \vdash' e : \pi \qquad \mathsf{U\,m}\,(\mathsf{U}_1,...,\mathsf{U}_n) = \mathit{choose}(\mathit{mSig}(\pi),\mathsf{m}) \qquad \forall i \in 1..n, \quad \Sigma \vdash' e_i : \pi_i \quad \pi_i <: \mathsf{U}_i}{\Sigma \vdash' e.\mathsf{m}(e_1,...,e_n) : \mathsf{U}} \qquad \text{(RT-INVK)}$$

$$\frac{\mathit{fields}(\mathsf{C}) = \mathsf{U}_1\,\mathsf{f}_1;...;\mathsf{U}_n\,\mathsf{f}_n; \qquad \forall i \in 1..n, \quad \Sigma \vdash' e_i : \pi_i \quad \pi_i <: \mathsf{U}_i}{\Sigma \vdash' \textbf{new}\ \mathsf{C}(e_1,...,e_n) : \mathsf{C}} \qquad \text{(RT-NEW)}$$

$$\frac{\Sigma \vdash' e : \pi \qquad \exists\ \mathsf{I}\{\mathsf{R}\}, \quad \pi <: \mathsf{I}\{\mathsf{R}\} \quad \vdash \mathsf{T} \lessdot \mathsf{R}\ \ \mathrm{OK}}{\Sigma \vdash' e\{\mathsf{T}\} : \pi} \qquad \text{(RT-REPL)}$$

Well-formed heap:

$$\frac{\begin{array}{l} \mathrm{Dom}(\mathscr{H}) = \mathrm{Dom}(\Sigma) \\ \forall \iota \in \mathrm{Dom}(\mathscr{H}), \quad \mathscr{H}(\iota) = \langle \mathsf{C},\ \mathsf{f}_1 : \iota_1, \ldots, \mathsf{f}_n : \iota_n,\ \overline{\mathsf{M}} \rangle \quad \textbf{implies} \\ \qquad \Sigma(\iota) = \mathsf{C} \qquad \mathit{fields}(\mathsf{C}) = \mathsf{U}_1\,\mathsf{f}_1;...;\mathsf{U}_n\,\mathsf{f}_n; \\ \qquad \forall i \in 1..n, \quad \Sigma(\iota_i) <: \mathsf{U}_i \\ \qquad \mathit{mSig}(\overline{\mathsf{M}}) \subseteq \mathit{mSig}(\mathsf{C}) \\ \qquad \forall\ \mathsf{V\ m}\ (\overline{\mathsf{V}}\,\overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \overline{\mathsf{M}}, \quad \exists\ \mathsf{V}', \quad \mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{V}} \vdash' \mathsf{e} : \mathsf{V}' \quad \mathsf{V}' <: \mathsf{V} \end{array}}{\Sigma \Vdash \mathscr{H}} \qquad \text{(WF-HEAP)}$$

Figure 9: FDTJ: Typing rules for runtime expressions and heaps

typing judgement does not contain constraints (as we will see, in order to prove the type soundness result, it is not necessary to infer constraints on runtime expressions or on methods introduced by means of dynamic replacement).

The associated typing rules, given at the top of Figure 9, make use of the subtyping judgement $\pi_1 <: \pi_2$ introduced in Section 4 (Figure 3). As for the typing rules for source language expressions (in Figure 5) also the typing rules for runtime expressions are syntax directed, with one rule for each form of expression. Note that, in rule (RT-FIELD), the expression e can be either this or ι (this is enforced by the first premise of the rule). The most interesting rule is the one for dynamic trait replacement, (RT-REPL). It relies on the fact that, if the runtime expression $e\{\mathsf{T}\}$ has been generated starting from a well-typed program (see Definition 4.16), then the type inferred for e must be a subtype of some source program type $\mathsf{I}\{\mathsf{R}\}$ such that the validity of the judgement $\vdash \mathsf{T} \lessdot \mathsf{R}\ \ \mathrm{OK}$ has been already established (at compile-time) when typing the source program (see Theorem 5.2).

The judgment for well formed heap has the form $\boxed{\Sigma \Vdash \mathscr{H}}$, read "heap $\mathscr{H}$ is well formed with respect to the environment Σ". The associated rule, given at the bottom of Figure 9, ensures that the environment Σ maps all the addresses defined in the heap into the type of the corresponding objects. It also ensures that, for every object stored in the heap, the fields of the object contain appropriate values and the methods of the object introduced by dynamic replacements are well-typed.

The FDTJ type soundness result (Theorem 5.1) comes in two parts: first the notion of well-typed program (see Definition 4.16) is related with the typing of runtime expressions (Theorem 5.2), then the type soundness for runtime expressions is given (Theorem 5.5). Theorem 5.2 can be proved by straightforward induction on the type derivation, while Theorem 5.5 follows immediately from subject reduction and progress (Theorems 5.3 and 5.4 — the proofs are given in Appendix A).

Theorem 5.2 ($\vdash$-typed closed expressions are $\vdash'$-typed). *If* $\bullet \vdash \mathsf{e} : \pi \mid \langle \bullet \mid \bullet \mid \bullet \rangle \mid \Delta$ *and* $\vdash \delta\ \ \mathrm{OK}$ *for all* $\delta \in \Delta$, *then* $\bullet \vdash' \mathsf{e} : \pi$.

Theorem 5.3 (Subject reduction). *If* $\Sigma \Vdash \mathscr{H}$, $\Sigma \vdash' e : \pi$ *and* $e, \mathscr{H} \longrightarrow e', \mathscr{H}'$, *then there exists* $\Sigma' \supseteq \Sigma$ *such that* $\Sigma' \Vdash \mathscr{H}'$, $\Sigma' \vdash' e' : \pi'$, *for some* π' *such that* $\pi' <: \pi$.

Theorem 5.4 (Progress). *Let* $\Sigma \Vdash \mathscr{H}$ *and* $\Sigma \vdash' e : \pi$, *where e is a runtime expression. Then either e is a value or there exist* e' *and* $\mathscr{H}'$ *such that* $e, \mathscr{H} \longrightarrow e', \mathscr{H}'$.

Theorem 5.5 (Type Soundness). *If* $\bullet \vdash' \mathsf{e}_0 : \pi$ *and* $\mathsf{e}_0, \emptyset \rightarrow^{\star} e, \mathscr{H}$ *with e a normal form, then e is an address* ι *and there exist* Σ *and* C *such that* $\Sigma \Vdash \mathscr{H}$, $\Sigma \vdash' \iota : \mathsf{C}$ *and* $\mathsf{C} <: \pi$.

6. Related Work

Some of the literature related to traits and dynamic trait replacement has already been quoted through the paper. Here we present a more detailed comparison between FDTJ and *Chai*3, and briefly discuss some other languages and calculi that provide type systems integrating nominal and structural subtyping or mechanisms for changing the behavior of objects at runtime.

The language *Chai*3 [41] does not have interfaces and therefore does not consider the issue of performing dynamic trait replacement when the static type of the target expression is an interface, that we have addressed by introducing types of the shape I{R}. The trait replacement operation of *Chai*3 must be applied to expressions whose static type is a class. The trait to be replaced must correspond to a trait T that was used in the class implementation and to replace a trait in an object it is necessary to specify such name. In particular, it is not possible to write a trait replacement operation that replaces a proper subset of the methods of T. For instance, the trait replacement operations (1), (2), (3) and (4) in Section 3.2 cannot be written in *Chai*3 since each of them replaces a proper subset of the methods in the trait TFifo used in the definition of the class Sequence in Listing 5. In some cases, the replacement of a proper subset of the methods of a trait used in the definition of a class can be encoded by replacing more methods. For instance, let TLo be the trait obtained from TLoExtractionPolicy by adding the methods in and isEmpty of trait TFifo, then the following *Chai*3 method changePolicy2 would behave like the method changePolicy1 in Section 3.3

```
void changePolicy2(Sequence s) {
   s<TFifo −> TLo>;  // (1”)
   System.out.println(s.get());
   s.out();
   s<TFifo −> TFifo>  // (2”)
}
```

However, while the method changePolicy1 in Section 3.3 works for both the versions of the class Sequence in Listings 5 and 8, the *Chai*3 method changePolicy2 works only for the versions of the class Sequence in Listing 5. In fact, in *Chai*3, when replacing a trait in an object it is necessary to specify the name of the trait used in the object's class definition. Therefore, modifying a class definition by changing the name of one of the traits used in the class implementation invalidates all the trait replacement operations associated to the changed trait. For instance, if the class Sequence in Listing 5 is changed as in Listing 8, so that it uses trait TLifo (i.e., sequences would be created as Last-in-first-out sequences), the code for the trait replacement operations (1”) and (2”) above should be changed into s<TLifo −> TLo> and s<TLifo −> TFifo>, respectively. Moreover, by replacing more methods, it is not possible to encode the methods disableIn and enableIn of Section 3.2 that accept as argument a sequence object and disable/enable the method in without affecting the methods out and get. In order to encode the methods disableIn and enableIn into *Chai*3, it is needed to consider a different strategy of encoding: for instance, by rewriting all the traits to contain just one provided method each, and then by expressing all the dynamic updates one method at time.

Dynamic method replacement is a basic feature of the Abadi-Cardelli imperative object calculus [1], a formalism aiming to encode (as many as possible) features of object-oriented languages. In this paper we have a different aim, that is, to formalize dynamic trait replacement within a JAVA-like nominal type system in order to foster its adoption (together with traits) in mainstream programming languages.

The FDTJ type system has a flavor of both structural and nominal systems. Integration of nominal and structural subtyping has recently received a considerable deal of interest (see, e.g., [29, 22, 34]). The shape of FDTJ source language types, I{R}, is similar to the shape of the types of Unity [29], $\beta(\overline{m:\tau})$ where β is a *brand* name and $\overline{m:\tau}$ is a sequence of method signatures. However, there are crucial differences. Brands are closer to JAVA classes rather than to JAVA interfaces: a brand declaration may contain the code of methods and the sub-branding inheritance hierarchy must be a tree. Moreover, Unity does not address issues of implementation inheritance, while traits provide a solution to these issues, and Unity (which is a pure functional calculus) is not able to (and does not aim to) express dynamic method replacement.

In [19] a dynamic inheritance mechanism for dynamic specialization of objects, is presented for the GBETA language. The proposed mechanism is quite flexible, however it is not type safe. Moreover, it relies on the submethoding feature supported by BETA, therefore its integration into JAVA-like languages would first require the integration of the submethoding feature.

Primitives for changing at runtime the class membership of an object are present, for instance, in the dynamic languages SMALLTALK and CLOS. The paper [17] presents, through the *Fickle*$_{\text{II}}$, language features for changing at runtime the class membership of an object within a JAVA-like nominal type system. In *Fickle*$_{\text{II}}$ only objects belonging to special classes, called *root* and *state* classes, can be reclassified and the type system restricts the use of these classes (in particular, state classes may not be used as types for fields). This makes the flexibility/expressivity of *Fickle*$_{\text{II}}$ similar to the one of *Chai*$_3$ [41]. We refer to [17] for a brief overview of approaches related to dynamic object reclassification. The *Fickle*$_3$ calculus [15] eliminates the need to declare explicitly the classes of the objects that may be re-classified. Re-classification may be decided by the client of a class, allowing unanticipated object re-classification. Still, the type system restricts the use of the classes of the objects that may be re-classified. In particular, let us consider classes C, C' and C'' where C' is a subclass of C while C'' is not. If there are objects belonging to C' that may be re-classified to C'' then C may not be used as type for fields. More recently, *typestate-oriented programming* [2, 38], a proposal very similar in spirit to the *Fickle*$_{\text{II}}$/*Fickle*$_3$ proposal, has overcome some of the limitations in *Fickle*$_{\text{II}}$/*Fickle*$_3$, e.g., the inability to track the states of fields. Both dynamic object reclassification and typestate-oriented programming rely on standard class-based inheritance, while the replaceable construct has been designed to work synergically with trait composition.

There are some similarities between our trait-based dynamic method replacement and approaches based on *delegation* [26, 43], which rely on object composition and method delegation as a more flexible and runtime version of class inheritance and method overriding: every object has a list of *parent* objects and when an object cannot answer a message it forwards it to its parents until there is an object that can process the message. However, a drawback of delegation is that runtime type errors ("message-not-understood") can arise when no delegates are able to process the forwarded message [44]. For this reason, some linguistic approaches were studied to deal with with delegation and type safety properties; we refer to Kniesel [25] for an overview of problems when combining delegation with a static type discipline.

Note that sometime, the term delegation is used with the simpler concept of method forward or *consultation* (see, e.g., [21, 11]). With delegation, when *A* delegates to *B* the execution of a message *m*, this is bound to the sender (*A*), thus, if in the body of the method *m* (defined in *B*) there is a call to a method *n*, then also this call will be executed binding this to *A*; while with consultation, during the execution of the body, the implicit parameter is always bound to the receiver *B*. Delegation is more powerful as it enables *dynamic method redefinition*. On the contrary, with consultation we lose the *transparent redirection* [35]; when we manually implement object composition and method forwarding we will not achieve a real dynamic object inheritance and dynamic method redefinition. For instance, when implementing *decorator* pattern [21], we will surely experience the anomaly known as *self problem* [26], i.e., *broken delegation* [23]. In the following we briefly describe some approaches that propose linguistic mechanisms based on delegation in a statically-typed setting.

In [33] a model based on *delegation layers* is presented where all the features that are typical of class-based languages (inheritance, delegation, late binding and subtype polymorphism) automatically apply to sets of collaborating classes and objects. In [35] *compound references* are introduced, a new abstraction for object references, which provides explicit linguistic support for combining different composition properties on-demand. The model is statically typed and allows the programmer to express several kinds of composition semantics in the interval between object composition and inheritance. More recently, in [5, 6] an extension of FJ with object composition and delegation is presented. In that calculus methods can be changed dynamically at runtime on an existing object as a consequence of object composition and "redefining" methods (a runtime version of standard method overriding).

We share with the above works on delegation the view that new fine grained linguistic features, that can be combined, can increase the flexibility in object oriented languages thus avoiding the need of many design patterns originally proposed to overcome lack of adequate constructs in a language. However, delegation based approaches rely on object-based features, mainly object composition (to this aim classes defining delegator objects usually have an explicit reference to the delegate object as in the State pattern), which are distant from our setting, especially because instead of relying on class inheritance, we are based on traits as the main mechanism for code reuse. In particular we focus on the unanticipated aspects that we detailed throughout the paper, so that we can smoothly change the behavior of existing objects in a safe way (using the concept of replaceable as the main contract ensuring type safety). Summarizing, in our setting of dynamic trait replacement, method substitutions are unanticipated in the sense that we do not have to structure the code in advance using composition of objects to be able to change subsets of methods. Note that in our approach we also achieve *transparent redirection* [35], thus avoid the before mentioned *self problem* [26] and *broken delegation* [23] problems.

It would be interesting, and subject of future work, to investigate how the approaches based on delegation could be merged into our dynamic trait settings, in particular, to study whether the trait approach, as opposed to class-based inheritance, could be a good setting for delegation and object composition.

7. Conclusion and Future Work

We proposed statically typed language features that decouple trait replacement operation code and class declaration code. These features support to refactor classes and/or performing unanticipated trait replacement operations without invalidating existing code. We illustrated these features through examples and proved the soundness of the associated typing rules by means of the FDTJ calculus.

Further work includes developing a prototypical implementation and validating the usability of the approach through examples. A possible implementation could consist in a preprocessor that generates JAVA code that relies on JAVA reflection, for selecting dynamically the replaced methods. Note that reflection might also be used to achieve method replacement manually, but in that case, type safety could not be checked statically.

We are also planning to strengthen the integration between nominal and structural subtyping and to add generics. Finally, it could be interesting to investigate method replacement in connection with classes/methods having invariants/contracts.

Acknowledgments. We thank the anonymous SCP referees and the referees of earlier versions of these paper for insightful comments and suggestions to improve the presentation.

References

[1] M. Abadi and L. Cardelli. An imperative object calculus. In *TAPSOFT*, volume 915 of *LNCS*, pages 471–485. Springer, 1995.

[2] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *OOPSLA*, pages 1015–1022. ACM, 2009.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, G.-W.Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstad. The Fortress Language Specification, Version 1.0, 2008.

[4] D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, 2003.

[5] L. Bettini and V. Bono. Type Safe Dynamic Object Delegation in Class-based Languages. In *PPPJ*, pages 171–180. ACM Press, 2008.

[6] L. Bettini, V. Bono, and B. Venneri. Delegation by object composition. *Science of Computer Programming*, 76(11):992–1014, 2011.

[7] L. Bettini, S. Capecchi, and F. Damiani. A Mechanisms for Flexible Dynamic Trait Replacement. In *FTfJP (http://www.cs.ru.nl/ftfjp/)*. ACM Digital Library, 2009.

[8] L. Bettini, S. Capecchi, and B. Venneri. Featherweight Java with Dynamic and Static Overloading. *Science of Computer Programming*, 74(5-6):261–278, 2009.

[9] V. Bono, F. Damiani, and E. Giachino. On Traits and Types in a Java-like setting. In *TCS 2008 (Track B)*, volume 273 of *IFIP*, pages 367–382. Springer, 2008.

[10] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA*, volume 25(10) of *SIGPLAN Notices*, pages 303–311. ACM Press, 1990.

[11] M. Büchi and W. Weck. Generic wrappers. In *ECOOP*, volume 1850 of *LNCS*, pages 201–225. Springer, 2000.

[12] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, 1995.

[13] C. Chambers, B. Harrison, and J. Vlissides. A debate on language and tool support for design patterns. In *POPL*, pages 277–289. ACM Press, 2000.

[14] C. Clifton, G. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. *ACM SIGPLAN Notices*, 35(10):130–145, 2000.

[15] F. Damiani, S. Drossopoulou, and P. Giannini. Refined effects for unanticipated object re-classification: Fickle3 (extended abstract). In *ICTCS*, volume 2841 of *LNCS*, pages 97–110. Springer, 2003.

[16] L. DeMichiel and R. Gabriel. The Common Lisp Object System: An Overview. In *ECOOP*, volume 276 of *LNCS*, pages 151–170. Springer, 1987.

[17] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. More dynamic object re-classification: Fickle$_{II}$. *ACM TOPLAS*, 24(2):153–191, 2002.

[18] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[19] E. Ernst. Dynamic inheritance in a statically typed language. *Nordic J. of Computing*, 6(1):72–92, 1999.

[20] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *POPL*, pages 171–183. ACM Press, 1998.

[21] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[22] J. Y. Gil and I. Maman. Whiteoak: Introducing Structural Typing into Java. In *OOPSLA*, pages 73–90. ACM, 2008.

[23] W. Harrison, H. Ossher, and P. Tarr. Using delegation for software and subject composition. Technical Report RC 20946, IBM Thomas J. Watson Research Center, Aug. 1997.

[24] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[25] G. Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In *ECOOP*, volume 1628 of *LNCS*, pages 351–366. Springer, 1999.

[26] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–214, 1986.

[27] M. V. Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[28] L. Liquori and A. Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM TOPLAS*, 30(2), 2008.

[29] D. Malayeri and J. Aldrich. Integrating Nominal and Structural Subtyping. In *ECOOP*, volume 5142 of *LNCS*, pages 260–284. Springer, 2008.

[30] W. Mugridge, J. Hamer, and J. Hosking. Multi-Methods in a Statically-Typed Programming Language. In *ECOOP*, volume 512 of *LNCS*, pages 307–324. Springer, 1991.

[31] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT*, 5(4):129–148, 2006.

[32] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.

[33] K. Ostermann. Dynamically composable collaborations with delegation layers. In *ECOOP*, volume 2374 of *LNCS*, pages 89–110. Springer, 2002.

[34] K. Ostermann. Nominal and structural subtyping in component-based programming. *JOT*, 7(1):121 – 145, 2008.

[35] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *OOPSLA*, pages 283–299. ACM, 2001.

[36] J. Reppy and A. Turon. A Foundation for Trait-based Metaprogramming. In *FOOL/WOOD*, 2006.

[37] J. Reppy and A. Turon. Metaprogramming with Traits. In *ECOOP*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.

[38] D. Saini, J. Sunshine, and J. Aldrich. A theory of typestate-oriented programming. In *Proceedings of the 12th Workshop on Formal Techniques for Java-Like Programs*, FTFJP '10, pages 9:1–9:7. ACM, 2010.

[39] L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *ECOOP*, volume 3586 of *LNCS*, pages 312–336. Springer, 2005.

[40] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable Units of Behavior. In *ECOOP*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.

[41] C. Smith and S. Drossopoulou. *Chai*: Traits for Java-Like Languages. In *ECOOP 2005*, LNCS 3586, pages 453–478. Springer, 2005.

[42] D. Ungar, C. Chambers, B. Chang, and U. Hölzle. Organizing programs without classes. *Lisp Symb. Comput.*, 4(3):223–242, 1991.

[43] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, pages 227–242. ACM Press, 1987.

[44] J. Viega, B. Tutt, and R. Behrends. Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages. Technical Report CS-98-03, UVa Computer Science, 1998.

A. Proofs of Theorems 5.3 and 5.4

Definition A.1. *We define the following auxiliary functions on trait expressions:*

- $\mathit{mRequired}(\mathsf{T}) = \mathit{mRequired}(\mathsf{TE})$ **if trait** T **is** TE
  $\mathit{mRequired}(\{\cdots;\ \overline{\mathsf{S}};\ \ldots\}) = \overline{\mathsf{S}}$
- $\mathit{fRequired}(\mathsf{T}) = \mathit{fRequired}(\mathsf{TE})$ **if trait** T **is** TE
  $\mathit{fRequired}(\{\overline{\mathsf{F}};\ \ldots;\ \ldots\}) = \overline{\mathsf{F}}$

A.1. Proof of Theorem 5.3

Remember that the rules for system $\vdash'$ (Figure 9) are syntax directed.

In order to prove Theorem 5.3 we first prove some auxiliary lemmas: Lemma A.2 (weakening), Lemma A.3 (substitution), Lemmas A.4 and A.5 (used to deal with method invocation) and Lemmas A.6 and A.7 (used to deal with trait replacement).

Lemma A.2 (Weakening). *If* $\Sigma \vdash' e : \pi$*, then* $\Sigma, \iota : \mathsf{C} \vdash' e : \pi$*.*

PROOF. Straightforward induction on the derivation of $\Sigma \vdash' e : \pi$. □

□

Lemma A.3 (Substitution). *If* $\Sigma, \overline{\mathsf{x}} : \overline{\pi} \vdash' e : \pi$ *and* $\Sigma \vdash' \overline{\mathsf{v}} : \overline{\mathsf{C}}$*, where* $\overline{\mathsf{C}} <: \overline{\pi}$*, then* $\Sigma \vdash' e[\overline{\mathsf{v}}/\overline{\mathsf{x}}] : \pi'$*, for some* $\pi' <: \pi$*.*

PROOF. By induction on the derivation of $\Sigma, \overline{\mathsf{x}} : \overline{\pi} \vdash' e : \pi$. We show only the most interesting cases.

Case $e_0.\mathbf{m}(\overline{e})$**.** Then $\Sigma, \overline{\mathsf{x}} : \overline{\pi} \vdash' e_0.\mathsf{m}(\overline{e}) : \pi$, where $\pi = \mathsf{U}$, and

$$\begin{array}{l} \Sigma \vdash' e_0 : \pi'_0 \qquad \forall i \in 1..n, \quad \Sigma \vdash' e_i : \pi'_i \quad \pi'_i <: \mathsf{U}_i \\ \mathsf{U}\,\mathsf{m}\,(\mathsf{U}_1,...,\mathsf{U}_n) = choose(mSig(\pi'_0), \mathsf{m}) \end{array}$$

for some U, π'_0, π'_1, $\ldots$, π'_n,U_1, $\ldots$,U_n. By induction we have $\forall i \in 0..n, \Sigma \vdash' e_i[\overline{\mathsf{v}}/\overline{\mathsf{x}}] : \pi''_i$ for some π''_i such that $\pi''_i <: \pi'_i$. By the subtyping relation, $mSig(\pi'_0) \subseteq mSig(\pi''_0)$. Therefore, $\mathsf{U}\,\mathsf{m}\,(\mathsf{U}_1,...,\mathsf{U}_n) \in choose(mSig(\pi''_0), \mathsf{m})$. Applying rule (RT-INVK) we get $\Sigma \vdash' (e_0.\mathsf{m}(\overline{e}))[\overline{\mathsf{v}}/\overline{\mathsf{x}}] : \mathsf{U}$ which proves the result.

Case new C$(\overline{e})$**.** Similar to the previous one.

Case $e_0\{\mathbf{T}\}$**.** Then $\Sigma \vdash' e_0\{\mathsf{T}\} : \pi$ and

$$\Sigma \vdash' e_0 : \pi \qquad \exists\, \mathsf{I}\{\mathsf{R}\}, \quad \pi <: \mathsf{I}\{\mathsf{R}\} \quad \vdash \mathsf{T} \lessdot \mathsf{R} \;\; \text{OK}$$

By induction $\Sigma \vdash' e_0[\overline{\mathsf{v}}/\overline{\mathsf{x}}] : \pi'$ for some π' such that $\pi' <: \pi$ and by transitivity $\pi' <: \pi <: \mathsf{I}\{\mathsf{R}\}$. Applying rule (RT-REPL) we get $\Sigma \vdash' e[\overline{\mathsf{v}}/\overline{\mathsf{x}}] : \pi'$ which proves the result. □

□

Lemma A.4. *If*

- $\vdash$ **class** C **implements** $\overline{\mathsf{I}}$ **by** TE $\{\ \overline{\mathsf{V}}\,\overline{\mathsf{g}};\ \ \mathsf{C}(\overline{\mathsf{V}}\,\overline{\mathsf{g}})\ \{\ldots\}\ \}$ OK
- *fRequired*(TE)= $\overline{\mathsf{F}}$ *and methods*(TE)=$\overline{\mathsf{M}}$
- $\mathsf{this} : \langle\ \overline{\mathsf{F}} \mid mSig(\overline{\mathsf{M}})\ \rangle, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash \mathsf{e} : \theta \mid \gamma \mid \Delta$

then $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash' \mathsf{e} : \pi$ *for some* π *such that:*
if $\theta = \langle\ \overline{\mathsf{F}} \mid mSig(\overline{\mathsf{M}})\ \rangle$ *then* $\pi = \mathsf{C}$*, else* $\pi <: \theta$*.*

PROOF. Assume (without loss of generality) that TE is a basic trait expression. By rules (C-OK) and (T-TEBASIC) we have $mRequired(\mathsf{TE}) = \bullet$. Then the result follows by straightforward induction on the derivation
$\mathsf{this} : \langle\ \overline{\mathsf{F}} \mid mSig(\overline{\mathsf{M}})\ \rangle, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash \mathsf{e} : \theta \mid \gamma \mid \Delta$. □

□

Lemma A.5. *If*

- $\vdash$ **class** C **implements** $\overline{\mathsf{I}}$ **by** TE$\{\ \overline{\mathsf{V}}\,\overline{\mathsf{g}};\ \ \mathsf{C}(\overline{\mathsf{V}}\,\overline{\mathsf{g}})\ \{\ldots\}\}$ OK
- W m $(\overline{\mathsf{W}}\,\overline{\mathsf{x}})\{$**return** e;$\} \in methods(\mathsf{TE})$

then $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash' \mathsf{e} : \pi$ *for some* π *such that* $\pi <: \mathsf{W}$*.*

PROOF. Assume (without loss of generality) that TE is a basic trait expression $\{\overline{\mathsf{F}};\ \overline{\mathsf{S}};\ \overline{\mathsf{M}}\}$ then $methods(\mathsf{TE}) = \overline{\mathsf{M}}$. By rules (C-OK) and (T-TEBASIC) we have $\overline{\mathsf{S}} = \bullet$. From hypothesis and rule (C-OK) we have

$$\begin{array}{ll} \vdash \{\overline{\mathsf{F}};\ \bullet;\ \overline{\mathsf{M}}\} : \mu_1 ... \mu_p \qquad p \geq 0 & \\ \forall i \in 1..p, \quad \mu_i = \zeta_i \mid \langle\ \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{U}}^{(i)}\ \rangle \mid \Delta^{(i)} & \\ \overline{\mathsf{V}}\,\overline{\mathsf{g}} \supseteq (\cup_{i \in 1..p} \overline{\mathsf{F}}^{(i)}) \cup (\cup_{i \in 1..p} fields(\overline{\mathsf{U}}^{(i)})) & (*) \\ \zeta_1 ... \zeta_p \supseteq (\cup_{i \in 1..p} \overline{\sigma}^{(i)}) \cup (\cup_{i \in 1..p} mSig(\overline{\mathsf{U}}^{(i)})) \cup mSig(\overline{\mathsf{I}}) & (**) \\ \forall \mathsf{I}' \in \cup_{i \in 1..p} allInterfaces(\overline{\mathsf{U}}^{(i)}), \quad \exists \mathsf{I} \in \overline{\mathsf{I}}, \quad \mathsf{I} \trianglelefteq \mathsf{I}' & (***) \end{array}$$

Let $\zeta_j = \mathsf{W}\ \mathsf{m}\ (\overline{\mathsf{W}})$ for some $j \in 1..p$. From rule (T-TEBASIC) we have

$$\begin{array}{lll} mSig(\overline{\mathsf{S}}) = \bullet & mSig(\mathsf{M}_1...\mathsf{M}_p) = \zeta_1...\zeta_p & p \geq 0 \\ \multicolumn{3}{l}{\mathsf{this} : \langle\ \overline{\mathsf{F}} \mid \zeta_1...\zeta_p\ \rangle \vdash \mathsf{W}\ \mathsf{m}\ (\overline{\mathsf{W}\,\mathsf{x}})\ \{\ \textbf{return}\ \mathsf{e};\ \} : \mu_j} \\ \multicolumn{3}{l}{\mu_j = \zeta_j \mid \langle\ \overline{\mathsf{F}}^{(j)} \mid \overline{\zeta}^{(j)} \mid \overline{\mathsf{U}}^{(j)}\ \rangle \mid \Delta^{(j)}} \end{array}$$

From rule (M-OK) we have

$$\mathsf{this} : \langle\ \overline{\mathsf{F}} \mid \zeta_1...\zeta_p\ \rangle, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash \mathsf{e} : \theta \mid \langle\ \overline{\mathsf{F}}^{(j)} \mid \overline{\sigma}^{(j)} \mid \overline{\mathsf{U}}'^{(j)}\ \rangle \mid \Delta^{(j)}$$

where:

- if $\theta = \langle\ \overline{\mathsf{F}} \mid \zeta_1...\zeta_p\ \rangle$ then $\overline{\mathsf{U}}^{(j)} = \overline{\mathsf{U}}'^{(j)} \cup \mathsf{W}$. Then by Lemma A.4, we have $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash' \mathsf{e} : \mathsf{C}$. Let $\mathsf{W} = \mathsf{I}_0\{\mathsf{R}_0\}$:
  - from (***) $\exists \mathsf{I} \in \overline{\mathsf{I}}, \quad \mathsf{I} \trianglelefteq \mathsf{I}_0$
  - by definition of R_C and (*), (**), (***) we have $\mathsf{R}_0 \sqsubseteq \mathsf{R}_\mathsf{C}$

  then by definition of subtyping we have $\mathsf{C} <: \mathsf{W}$.
- if $\theta \neq \langle\ \overline{\mathsf{F}} \mid \zeta_1...\zeta_p\ \rangle$ then $\theta <: \mathsf{W}$ and $\overline{\mathsf{U}}^{(j)} = \overline{\mathsf{U}}'^{(j)}$.

  Then by Lemma A.4, we have $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash' \mathsf{e} : \pi$ with $\pi <: \theta <: \mathsf{W}$. □

□

Lemma A.6. *If*

- $\vdash\ \ \textbf{class}\ \mathsf{C}\ \textbf{implements}\ \overline{\mathsf{I}}\ \textbf{by}\ \mathsf{TE}\ \ \{\ \ \overline{\mathsf{V}\,\mathsf{g}};\ \ \mathsf{C}(\overline{\mathsf{V}\,\mathsf{g}})\ \{\ldots\}\ \ \}\ \ \text{OK}$
- $\vdash \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \mathsf{TE}' : \overline{\mu}$
- $\exists\ \mathsf{I}\{\mathsf{R}\}, \quad \mathsf{C} <: \mathsf{I}\{\mathsf{R}\} \quad \vdash \mathsf{T} \lessdot \mathsf{R}\ \ \text{OK}$
- *fRequired*$(\mathsf{T}) = \overline{\mathsf{F}}$, *mRequired*$(\mathsf{T}) = \overline{\mathsf{S}}$ *and methods*$(\mathsf{T}) = \overline{\mathsf{M}}$
- $\mathsf{this} : \langle\ \overline{\mathsf{F}} \mid mSig(\overline{\mathsf{M}}) \cup mSig(\overline{\mathsf{S}})\ \rangle, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash \mathsf{e} : \theta \mid \gamma \mid \Delta$

then $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash' \mathsf{e} : \pi$ *for some* π *such that:*
if $\theta = \langle\ \overline{\mathsf{F}} \mid mSig(\overline{\mathsf{M}}) \cup mSig(\overline{\mathsf{S}})\ \rangle$ *then* $\pi = \mathsf{C}$*, else* $\pi <: \theta$.

PROOF. By straightforward induction on the derivation
$\mathsf{this} : \langle\ \overline{\mathsf{F}} \mid mSig(\overline{\mathsf{M}}) \cup mSig(\overline{\mathsf{S}})\ \rangle, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash \mathsf{e} : \theta \mid \gamma \mid \Delta$. □

□

Lemma A.7. *If*

- $\vdash\ \ \textbf{class}\ \mathsf{C}\ \textbf{implements}\ \overline{\mathsf{I}}\ \textbf{by}\ \mathsf{TE}\ \ \{\ \ \overline{\mathsf{V}\,\mathsf{g}};\ \ \mathsf{C}(\overline{\mathsf{V}\,\mathsf{g}})\ \{\ldots\}\ \ \}\ \ \text{OK}$
- $\vdash \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \mathsf{TE}' : \overline{\mu}$
- $\exists\ \mathsf{I}\{\mathsf{R}\}, \quad \mathsf{C} <: \mathsf{I}\{\mathsf{R}\} \quad \vdash \mathsf{T} \lessdot \mathsf{R}\ \ \text{OK}$
- $\mathsf{W}\ \mathsf{m}\ (\overline{\mathsf{W}\,\mathsf{x}})\{\textbf{return}\ \mathsf{e};\} \in methods(\mathsf{T})$

then $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash' \mathsf{e} : \pi$ *for some* π *such that* $\pi <: \mathsf{W}$.

PROOF. Assume (without loss of generality) that TE is a basic trait expression $\{\overline{\mathsf{F}};\ \overline{\mathsf{S}};\ \overline{\mathsf{M}}\}$. By rules (C-OK) and (T-TEBASIC) we have $\overline{\mathsf{S}} = \bullet$. From hypothesis and rule (C-OK) we have

$$\begin{array}{l}
\vdash \{\overline{\mathsf{F}};\ \bullet;\ \overline{\mathsf{M}}\} : \mu_1 ... \mu_p \qquad\qquad p \geq 0 \\
\forall i \in 1..p, \quad \mu_i = \zeta_i \mid \langle\, \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{U}}^{(i)} \,\rangle \mid \Delta^{(i)} \\
\overline{\mathsf{V}}\ \overline{\mathsf{g}} \supseteq (\cup_{i\in 1..p}\overline{\mathsf{F}}^{(i)}) \cup (\cup_{i\in 1..p}\mathit{fields}(\overline{\mathsf{U}}^{(i)})) \\
\zeta_1 ... \zeta_p \supseteq (\cup_{i\in 1..p}\overline{\sigma}^{(i)}) \cup (\cup_{i\in 1..p}\mathit{mSig}(\overline{\mathsf{U}}^{(i)})) \cup \mathit{mSig}(\bar{\mathsf{I}}) \\
\forall \mathsf{I}' \in \cup_{i\in 1..p}\mathit{allInterfaces}(\overline{\mathsf{U}}^{(i)}), \quad \exists \mathsf{I} \in \bar{\mathsf{I}}, \quad \mathsf{I} \trianglelefteq \mathsf{I}'
\end{array}$$

Assume (without loss of generality) that TE' is a basic trait expression $\{\overline{\mathsf{F}}';\ \overline{\mathsf{S}}';\ \overline{\mathsf{M}}'\}$. From rule (T-TEBASIC) we have

$$\begin{array}{l}
\mathit{mSig}(\overline{\mathsf{S}}') = \overline{\sigma}' \qquad\quad \mathit{mSig}(\mathsf{M}'_1 ... \mathsf{M}'_{p'}) = \zeta'_1 ... \zeta'_{p'} \qquad\quad p' \geq 0 \\
\forall i \in 1..p' \ \ \mathsf{this} : \langle\, \overline{\mathsf{F}}' \mid \overline{\sigma}' \cdot \zeta'_1 ... \zeta'_{p'} \,\rangle \vdash \mathsf{W}\ \mathsf{m}\ (\overline{\mathsf{W}}\ \overline{\mathsf{x}})\ \{\ \textbf{return}\ \mathsf{e};\ \} : \mu'_i \\
\mu_i = \zeta'_j \mid \langle\, \overline{\mathsf{F}}'^{(i)} \mid \overline{\zeta}'^{(i)} \mid \overline{\mathsf{U}}'^{(i)} \,\rangle \mid \Delta'^{(i)}
\end{array}$$

Let $\zeta'_j = \mathsf{W}\ \mathsf{m}\ (\overline{\mathsf{W}})$ for some $j \in 1..p'$. From rule (M-OK) we have

$$\mathsf{this} : \langle\, \overline{\mathsf{F}}' \mid \overline{\sigma}' \cdot \zeta'_1 ... \zeta'_{p'} \,\rangle, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash \mathsf{e} : \theta \mid \langle\, \overline{\mathsf{F}}'^{(j)} \mid \overline{\sigma}'^{(j)} \mid \overline{\mathsf{U}}''^{(j)} \,\rangle \mid \Delta'^{(j)}$$

where:

- If $\theta = \langle\, \overline{\mathsf{F}}' \mid \overline{\sigma}' \cdot \zeta'_1 ... \zeta'_p \,\rangle$ then $\overline{\mathsf{U}}'^{(j)} = \mathsf{V}''^{(j)} \cup \mathsf{W}$. By Lemma A.6, we have $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash' \mathsf{e} : \mathsf{C}$. Let $\mathsf{W} = \mathsf{I}_0\{\mathsf{R}_0\}$. By definition of R_T we have $\mathsf{R}_0 \sqsubseteq \mathsf{R}_\mathsf{T}$. From $\vdash \mathsf{T} \lessdot \mathsf{R}$ OK by rule ($\lessdot$-OK) we have $\mathsf{R}_\mathsf{T} \sqsubseteq \mathsf{R}$. From $\mathsf{C} <: \mathsf{I}\{\mathsf{R}\}$ and subtyping rule we have $\mathsf{R} \sqsubseteq \mathsf{R}_\mathsf{C}$. Then:

    - By transitivity we have $\mathsf{R}_\mathsf{T} \sqsubseteq \mathsf{R}_\mathsf{C}$ and since $\mathsf{I}_0 \in \mathsf{R}_\mathsf{T}$ by replaceable inclusion rule we have $\exists \mathsf{I} \in \bar{\mathsf{I}} : \mathsf{I} \trianglelefteq \mathsf{I}_0$
    - By transitivity we also have $\mathsf{R}_0 \sqsubseteq \mathsf{R}_\mathsf{C}$

    Therefore by definition of subtyping we have $\mathsf{C} <: \mathsf{W}$

- If $\theta \neq \langle\, \overline{\mathsf{F}}' \mid \overline{\sigma}' \cdot \zeta'_1 ... \zeta'_p \,\rangle$ then $\theta <: \mathsf{W}$ and $\overline{\mathsf{U}}'^{(j)} = \overline{\mathsf{U}}$.

    Then by Lemma A.6, we have $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{W}} \vdash' \mathsf{e} : \pi$ with $\pi <: \theta <: \mathsf{W}$. □

□

Proof of Theorem 5.3 (Subject Reduction). The proof is by induction on a derivation of $e, \mathscr{H} \longrightarrow e', \mathscr{H}'$, with a case analysis on the reduction rule used. We show only the most interesting cases for computation rules; for congruence rules simply use the induction hypothesis.

Case (R-INVK). The last applied rule is

$$\iota.\mathsf{m}(\bar{\iota}), \mathscr{H} \longrightarrow \mathsf{e}[{}^{\iota}/_{\mathsf{this}}, {}^{\bar{\iota}}/_{\overline{\mathsf{x}}}], \mathscr{H}$$

where $\mathscr{H}(\iota) = \langle \mathsf{C},\ \bar{\mathsf{f}} : \bar{\iota},\ \overline{\mathsf{M}} \rangle$ and $\mathsf{U}\ \mathsf{m}\ (\overline{\mathsf{U}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \mathit{exclude}(\mathit{methods}(\mathsf{C}), \mathit{names}(\overline{\mathsf{M}})) \cup \overline{\mathsf{M}}$. From rule (RT-INVK) we have

$$\begin{array}{l}
\Sigma \vdash' \iota : \mathsf{C}' \qquad \forall i \in 1..n, \quad \Sigma \vdash' \iota_i : \mathsf{C}_i \quad \mathsf{C}_i <: \mathsf{U}_i \\
\mathsf{U}\,\mathsf{m}\,(\mathsf{U}_1, ..., \mathsf{U}_n) = \mathit{choose}(\mathit{mSig}(\mathsf{C}'), \mathsf{m})
\end{array}$$

where $\Sigma \Vdash \mathscr{H}$ implies $\mathsf{C}' = \mathsf{C}$. There are two cases:

1. $\mathsf{V}\ \mathsf{m}\ (\overline{\mathsf{V}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \overline{\mathsf{M}}$ then from $\Sigma \Vdash \mathscr{H}$ we have $\mathit{mSig}(\overline{\mathsf{M}}) \subseteq \mathit{mSig}(\mathsf{C})$ thus $\mathsf{V} = \mathsf{U}$, $\overline{\mathsf{V}} = \overline{\mathsf{U}}$, $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash' \mathsf{e} : \mathsf{V}'$ and $\mathsf{V}' <: \mathsf{U}$. Therefore, by Lemma A.3 and A.2, we have that $\Sigma \vdash' \mathsf{e}[{}^{\iota}/_{\mathsf{this}}, {}^{\bar{\iota}}/_{\overline{\mathsf{x}}}] : \pi$, where $\pi <: \mathsf{V}' <: \mathsf{U}$ which proves the result.

2. $\mathsf{U}\ \mathsf{m}\ (\overline{\mathsf{U}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \mathit{methods}(\mathsf{C})$. Since class C is well-formed, then method m is well-typed and then, from Lemma A.5, we have that $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{U}} \vdash' \mathsf{e} : \pi$, where $\pi <: \mathsf{U}$. Therefore, by Lemma A.3 and A.2, we have that $\Sigma \vdash' \mathsf{e}[\iota/\mathsf{this}, \overline{\iota}/\overline{\mathsf{x}}] : \pi'$, where $\pi' <: \pi <: \mathsf{U}$.

Case (R-NEW). We have that

$$\textbf{new}\ \mathsf{C}(\overline{\iota}), \mathscr{H} \longrightarrow \iota, \mathscr{H} \cup \{\iota \mapsto \langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota},\ \bullet\rangle\},$$

where $\iota \notin \mathrm{Dom}(\mathscr{H})$. From rule (RT-NEW) we have

$$\mathit{fields}(\mathsf{C}) = \mathsf{U}_1\,\mathsf{f}_1; \ldots; \mathsf{U}_n\,\mathsf{f}_n; \quad \forall i \in 1..n, \Sigma \vdash' e_i : \pi_i \quad \pi_i <: \mathsf{U}_i$$

Let Σ' be such that $\Sigma' = \Sigma\ \cup\ \{\iota : \mathsf{C}\}$. Then $\Sigma' \vdash \iota : \mathsf{C}$ and $\Sigma' \Vdash \mathscr{H}'$, where $\mathscr{H}' = \mathscr{H} \cup \{\iota \mapsto \langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota},\ \bullet\rangle\}$.

Case (R-REPL). We have that

$$\iota\{\mathsf{T}\}, \mathscr{H} \longrightarrow \iota, \mathscr{H}[\iota \mapsto \langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota},\ \mathit{exclude}(\overline{\mathsf{M}}, \mathit{names}(\overline{\mathsf{M}}')) \cup \overline{\mathsf{M}}'\rangle]$$

where $\mathscr{H}(\iota) = \langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota},\ \overline{\mathsf{M}}\rangle$ and $\mathit{methods}(\mathsf{T}) = \overline{\mathsf{M}}'$. From rule (RT-REPL) we have:

$$\Sigma \vdash' \iota : \mathsf{C}' \qquad \exists\ \mathsf{I}\{\mathsf{R}\}, \quad \mathsf{C}' <: \mathsf{I}\{\mathsf{R}\} \quad \vdash \mathsf{T} \lessdot \mathsf{R}\ \ \mathrm{OK}$$

where $\Sigma \Vdash \mathscr{H}$ implies $\mathsf{C}' = \mathsf{C}$, thus $\mathsf{C} <: \mathsf{I}\{\mathsf{R}\}$. To prove that $\Sigma \Vdash \mathscr{H}'$ we have to show that:

1. $\mathit{mSig}(\mathit{exclude}(\overline{\mathsf{M}}, \mathit{names}(\overline{\mathsf{M}}'))) \cup \overline{\mathsf{M}}' \subseteq \mathit{mSig}(\mathsf{C})$
2. $\forall\ \mathsf{V}\ \mathsf{m}\ (\overline{\mathsf{V}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \mathit{exclude}(\overline{\mathsf{M}}, \mathit{names}(\overline{\mathsf{M}}')) \cup \overline{\mathsf{M}}'\ \exists\ \mathsf{V}'$ such that $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{V}} \vdash' \mathsf{e} : \mathsf{V}'$ with $\mathsf{V}' <: \mathsf{V}$.

Point (1). By hypothesis we have $\mathit{mSig}(\overline{\mathsf{M}}) \subseteq \mathit{mSig}(\mathsf{C})$ we just have to show $\mathit{mSig}(\overline{\mathsf{M}}') \subseteq \mathit{mSig}(\mathsf{C})$. From rule ($\lessdot$-OK) and (T-TEBASIC) we have

$$\begin{array}{l}
\vdash \textbf{trait}\ \mathsf{T}\ \textbf{is}\ \mathsf{TE} : \mu_1 \ldots \mu_p \qquad\quad p \geq 0 \\
\forall i \in 1..p, \qquad \mu_i = \zeta_i \mid \langle\ \overline{\mathsf{F}}^{(i)} \mid \overline{\sigma}^{(i)} \mid \overline{\mathsf{U}}^{(i)}\ \rangle \mid \Delta^{(i)} \\
\textbf{replaceable}\ \mathsf{R}\ \textbf{is}\ \{\overline{\mathsf{S}}\}\langle \overline{\mathsf{G}} \mid \overline{\mathsf{Z}} \mid \overline{\mathsf{J}}\rangle \\
\mathsf{R}_\mathsf{T} \sqsubseteq \mathsf{R} \text{ which implies } \mathit{mSig}(\overline{\mathsf{S}}) \supseteq (\cup_{i \in 1..p} \zeta^{(i)})
\end{array}$$

thus $(\cup_{i \in 1..p} \zeta^{(i)}) = \mathit{mSig}(\overline{\mathsf{M}}') \subseteq \mathit{mSig}(\overline{\mathsf{S}})$. Let

$$\begin{array}{l}
\textbf{class}\ \mathsf{C}\ \textbf{implements}\ \overline{\mathsf{I}}\ \textbf{by}\ \mathsf{TE}'\ \{\ \overline{\mathsf{F}};\ \mathsf{K}\ \} \\
\overline{\mathsf{M}}'' = \mathit{methods}(\mathsf{TE}')
\end{array}$$

then from subtyping rule $\mathsf{C} <: \mathsf{I}\{\mathsf{R}\}$ we have that $\mathsf{R} \sqsubseteq \mathsf{R}_\mathsf{C}$ which implies

$$\mathit{mSig}(\overline{\mathsf{M}}'') \supseteq \mathit{mSig}(\overline{\mathsf{S}} \cup \overline{\mathsf{Z}})$$

Thus $\mathit{mSig}(\mathsf{C}) = \mathit{mSig}(\overline{\mathsf{M}}'') \supseteq \mathit{mSig}(\overline{\mathsf{S}} \cup \overline{\mathsf{Z}}) \supseteq \mathit{mSig}(\overline{\mathsf{S}})$ which proves the result.
Point (2). By hypothesis we have $\forall\ \mathsf{V}\ \mathsf{m}\ (\overline{\mathsf{V}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \overline{\mathsf{M}}, \exists\ \mathsf{V}'$ such that $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{V}} \vdash' \mathsf{e} : \mathsf{V}'$ with $\mathsf{V}' <: \mathsf{V}$. By Lemma A.7 we have $\forall\ \mathsf{V}\ \mathsf{m}\ (\overline{\mathsf{V}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e};\ \} \in \overline{\mathsf{M}}', \exists\ \mathsf{V}'$ s. t. $\mathsf{this} : \mathsf{C}, \overline{\mathsf{x}} : \overline{\mathsf{V}} \vdash' \mathsf{e} : \mathsf{V}'$ with $\mathsf{V}' <: \mathsf{V}$ which proves the result. □

A.2. Proof of Theorem 5.4

Lemma A.8. *Given the configuration $e, \mathscr{H}$ such that $\Sigma \Vdash \mathscr{H}$ and $\Sigma \vdash' e : \pi$, then*

1. *If $e = \iota.\mathsf{f}_i$ then $\mathscr{H}(\iota) = \langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota}, \ldots,\ \overline{\mathsf{M}}\rangle$ for some C, $\overline{\mathsf{f}}$, $\overline{\iota}$ and $\overline{\mathsf{M}}$ such that $\mathsf{f}_i \in \overline{\mathsf{f}}$;*
2. *If $e = \iota.\mathsf{m}(\overline{\iota}')$ then $\mathscr{H}(\iota) = \langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota},\ \overline{\mathsf{M}}\rangle$ for some C, $\overline{\mathsf{f}}$, $\overline{\iota}$ and $\overline{\mathsf{M}}$ such that $\mathsf{U}\ \mathsf{m}\ (\overline{\mathsf{U}}\ \overline{\mathsf{x}})\{\ \textbf{return}\ \mathsf{e}';\ \} \in \mathit{exclude}(\mathit{methods}(\mathsf{C}), \mathit{names}(\overline{\mathsf{M}})) \cup \overline{\mathsf{M}}$ and $|\overline{\mathsf{x}}| = |\overline{\iota}'|$*
3. *If $e = \textbf{new}\ \mathsf{C}(\overline{\iota})$ then $\mathit{fields}(\mathsf{C}) = \overline{\mathsf{U}}\ \overline{\mathsf{f}}$ for some $\overline{\mathsf{U}}$, $\overline{\mathsf{f}}$ and $|\overline{\mathsf{f}}| = |\overline{\iota}|$*
4. *If $e = \iota\{\mathsf{T}\}$ then $\mathscr{H}(\iota) = \langle \mathsf{C},\ \overline{\mathsf{f}} : \overline{\iota},\ \overline{\mathsf{M}}\rangle$ and $\mathit{methods}(\mathsf{T})$=$\overline{\mathsf{M}}$' for some C, $\overline{\mathsf{f}}$, $\overline{\iota}$, $\overline{\mathsf{M}}$ and $\overline{\mathsf{M}}$'.*

PROOF.

1. Follows directly from well-formedness of heap.
2. From rule (RT-INVK) we have that
   $\Sigma \vdash' \iota : \mathsf{C}' \qquad \forall i \in 1..n, \quad \Sigma \vdash' \iota_i' : \mathsf{C}_i \quad \mathsf{C}_i <: \mathsf{U}_i$
   $\mathsf{U}\,\mathsf{m}\,(\mathsf{U}_1, ..., \mathsf{U}_n) = \mathit{choose}(\mathit{mSig}(\mathsf{C}'), \mathsf{m})$
   where $\Sigma \Vdash \mathscr{H}$ implies C' = C
   There are two cases:
   (a) $\mathsf{U}'\ \mathsf{m}\ (\overline{\mathsf{U}}'\ \bar{\mathsf{x}})\{$ **return** $\mathsf{e}';\ \} \in \overline{\mathsf{M}}$ then from $\Sigma \Vdash \mathscr{H}$ we have $\mathit{mSig}(\overline{\mathsf{M}}) \subseteq \mathit{mSig}(\mathsf{C})$ and thus $\mathsf{U} = \mathsf{U}'$ and $\overline{\mathsf{U}} = \overline{\mathsf{U}}'$.
   (b) $\mathsf{U}\ \mathsf{m}\ (\overline{\mathsf{U}}\ \bar{\mathsf{x}})\{$ **return** $\mathsf{e}';\ \} \in \mathit{methods}(\mathsf{C})$. Since C is well typed it easy to verify that such definition exists, i.e. $\mathit{methods}(\mathsf{C}) = \mathit{methods}(\mathsf{TE})$ for some TE if **class** C **implements** $\bar{\mathsf{I}}$ **by** TE $\{\ \overline{\mathsf{F}};\ \mathsf{K}\ \}$.
   Note that $|\bar{\mathsf{x}}| = |\bar{\iota}|$ derives from rule (RT-INVK) since $|\bar{\iota}| = |\overline{\mathsf{U}}|$.
3. Follows directly from rule (RT-NEW).
4. From rule (RT-REPL) and well formedness of the heap we have $\mathscr{H}(\iota) = \langle \mathsf{C},\ \bar{\mathsf{f}} : \bar{\iota},\ \overline{\mathsf{M}} \rangle$ for some C, $\bar{\mathsf{f}}$, $\bar{\iota}$ and $\overline{\mathsf{M}}$. From rule (RT-REPL) we have $\vdash \mathsf{T} \lessdot \mathsf{R}\quad \mathrm{OK}$ then from rules ($\lessdot$-OK) and (T-OK) we have $\mathit{methods}(\mathsf{T}) = \overline{\mathsf{M}}'$ for some $\overline{\mathsf{M}}'$. □

□

Proof of Theorem 5.4 (Progress). The proof is by straightforward induction on the derivation of $\Sigma \vdash' e : \pi$ using Lemma A.8 in the basic cases. □